View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

© 2006 by Wensheng Wu. All rights reserved.

INTEGRATING DEEP WEB DATA SOURCES

BY

WENSHENG WU

B.E., Tongji University, 1991
M.S., Fudan University, 1994

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

A large number of data sources on the Web (e.g., Amazon.com) are only accessible through their query interfaces. These sources are commonly known as *Deep Web* sources. For any domain of interest, there may be many such sources with varied query capabilities and content coverage. As a result, users frequently need to access multiple sources in order to find the desired information, which can be a very time-consuming and labor-expensive process. To address this problem, an effective solution is to build a *virtual* integration system over the sources. Such a system provides uniform accesses to the sources, thus freeing the users from the details of individual sources.

As an important step towards this goal, this dissertation studies the problem of integrating query interfaces of Deep Web sources. Interface integration typically involves three very challenging tasks: (1) *schema extraction*, which infers the schema of each source query interface from its (HTML) representation; (2) *schema matching*, which accurately identifies semantic mappings among the attributes from different interfaces; and (3) *schema merging*, which properly merges the source interfaces into a well-formed global interface based on the identified attribute mappings.

This dissertation presents IceQ, a novel and effective interface integration system. In developing IceQ, we address the limitations of existing solutions and make several key contributions. First, we propose a hierarchical modeling of interfaces and develop a novel spatial clustering algorithm to extract the hierarchical schema of query interface. Second, we develop a novel interactive clustering-based matching algorithm to accurately match a large number of schemas and effectively resolve uncertain mappings via user interaction. Third, we develop a question-answering technique to learn attribute instances from the Web to assist in schema matching. Fourth, we propose a novel constraint-based optimization framework for merging schemas and develop an effective merging algorithm based on the idea of clustering aggregation. Extensive experiments have been conducted to evaluate IceQ and the results show that it is highly effective.

To my dear family.

Acknowledgments

I am indebted to many people who have supported me over the years. First, I am deeply grateful to my advisors AnHai Doan, Clement Yu, and Geneva Belford. Thank you for all your guidance and support. Without them, this work would not have been possible. I thank AnHai for introducing me to the data integration problem and guiding me throughout my thesis work, for teaching me how to write well and communicate effectively, and for giving me numerous invaluable advices on my research and tremendous helps on my job-hunting process. I thank Clement for showing me how to become a good researcher, for teaching me all I know about information retrieval, and for giving me so many insightful comments on both my research topic and paper writing. I thank Geneva for giving me guidance on both my research and life, for encouraging me, and for helping me make many important decisions of my life. AnHai, Clement, and Geneva, you are such great advisors and mentors, thank you!

I thank Professor Jiawei Han for the invaluable feedbacks, comments, and discussions on my research work. I thank Professor Weiyi Meng for giving me many lessons on paper writing, and for supporting me on my job-hunting. I also owe special debt to my undergraduate supervisor and mentor, Professor Lu Wan Qing, for her support and encouragement.

This work also greatly benefits from the comments and feedbacks from: Pedro DeRose, Yoonkyong Lee, Robert McCann, Mayssam Sayyadian, and Warren Shen. Thank you all, you are such terrific groupmates.

I am deeply grateful to my parents and my wife, for their love, support, and encouragement. This work is dedicated to you.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation studies the problem of building data integration systems over the Deep Web data sources. In particular, we focus on the problem of integrating query interfaces of Deep Web sources.

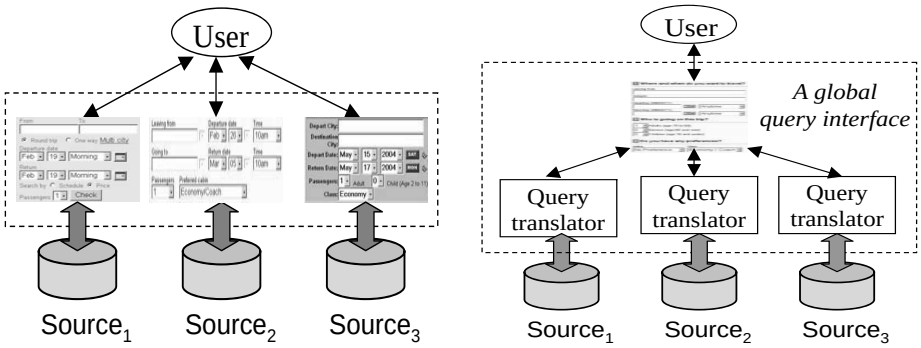
In this chapter, we start by discussing the search problem on the Deep Web and motivate the integration solution to the problem (Sections 1.1 & 1.2). We then describe the interface integration, its challenges and the limitations of existing solutions (Sections 1.3 & 1.4). Next, we introduce the proposed interface integration system IceQ and describe its key contributions (Section 1.5). Finally, we give an outline of the rest of the dissertation (Section 1.6).

1.1 Search Problem on the Deep Web

The Web is often divided into the Surface Web and the Deep Web [10, 57]. The Surface Web consists of billions of browsable pages, e.g., IBM home page. These pages are searchable from search engines such as Google. On the other hand, there are also a large number of pages which are hidden behind query interfaces and dynamically generated by data sources such as Amazon.com. These pages are typically not searchable from Google and form the so-called *Deep Web*. And the data sources which generate these dynamic pages are called Deep Web sources.

The Deep Web is becoming a very important information resource [10]. First, the Deep Web is quickly growing in the number of sources. In 2000, there are about 96,000 sources on the Deep Web. And the number has increased by 7 times in 2004 [15]. Second, the Deep Web contains a huge amount of valuable information. In fact, the number of pages on the Deep Web is estimated to be at least 500 times more than that on the Surface Web. Third, the Deep Web covers a very broad range of subjects, including business, entertainment, and government information.

But searching information on the Deep Web has become a very serious problem. For any domain of



(a) *Before* the interface integration (b) *After* the interface integration

Figure 1.1: Accessing the Deep Web sources

interest, there may be many sources which provide similar contents, but with varied coverage and querying capabilities. As a result, to find the desired information, users often need to interact with multiple sources, understand their query syntaxes, formulate separate queries, and compile query results from different sources. This can be an extremely time-consuming and labor-intensive process.

Example 1 Figure 1.1.a shows several sources (e.g., airliners and travel agents) which provide inquiry & booking services on the flights. Consider a user who is attending a conference and needs to find the lowest airfare for a round-trip flight from Chicago to New York with a price of under $500. Since different sources may provide different flights and offer varied discounts even on the same flight, the user may need to spend hours searching over multiple sources in order to find the desired flight. □

1.2 Deep Web Data Integration

To address this search problem on the Deep Web, an effective solution is to build a data integration system over a set of sources in a domain of interest. Such a system enables a unified access to the sources, thus freeing the users from the details of individual sources. To achieve this, the system first needs to provide the users with a *global query interface* so that the users only need to pose queries on the global query interface and the system will automatically translates the queries and pose them to different sources. The system also needs to combine all results from different sources and present them to the users on a *unified result interface*.

The Deep Web problem has received great attention from both research communities & industry. Many research communities, including database, Web, AI & KDD, are involved, with more than 10 research

groups from US, Asia & Europe. And the main focuses are source discovery, schema matching & integration. There are also many companies working on the Deep Web problem, e.g., Transformic, Glenbrook Networks, WebScalers, PriceGrabber, Shopping, MySimon and Google, to name just a few.

But despite these efforts, it is still very difficult to build and maintain data integration systems over the Deep Web sources. The reason is that it involves many challenging tasks.

- *Source discovery:* First, given a domain of interest, we need to be able to quickly discover existing sources on the Deep Web. Note that each Deep Web source has a Web page which contains its query interface written in HTML form. But the challenge is how to effectively search & identify these pages among billions of pages on the Web.

- *Schema extraction*: Each source query interface is associated with a schema which specifies query attributes as well as the grouping and ordering relationships among the attributes. But the schema is not explicitly given and needs to be extracted from the HTML representation of the query interface.

- *Schema matching*: Due to the autonomous nature of sources, different sources may represent similar attributes on their query interfaces very differently. So the challenge is how to accurately identify semantic correspondences among the attributes from different source interfaces. Such mappings are crucial to the interchange of data among different sources and also to the construction of global query interfaces (the next task).

- *Schema merging*: To enable uniform accesses to a set of sources, we need to provide users with a global query interface. The global query interface is typically constructed by merging source interfaces. As noted above, different sources may vary greatly in their ways of organizing attributes on their query interfaces. So the challenge is how to properly merge a large number of diversified source interfaces into a well-formed global interface.

- *Query optimization*: For each query users pose on the global query interface, we need to formulate effective and efficient query plans to execute the query over a large number of sources. We will discuss query optimization in more detail in Section 7.2.

- *Result merging*: After results are obtained from different sources, they first need to be properly combined (e.g. removing duplicates) before presented to the users. Result merging is an important future

direction and will be discussed in detail in Section 7.2.

- And finally, *system maintenance*: The sources may change constantly over time, e.g., new attributes may be added to their query interfaces and search results may be presented in a different format. So the challenge is how to effectively monitor these changes and automatically adjust the integration system on-the-fly.

1.3 Interface Integration

As an important step towards the integration of Deep Web sources, this dissertation focuses on the problem of integrating their query interfaces. In particular, given a set of sources in a domain of interest, we consider an interface integration system which enables the users to pose all the queries on a *global* query interface provided by the system. The system will then translate the queries into formats suitable for the sources, and automatically pose the queries to the sources, thus making the accesses to individual sources transparent to the users.

Example 2 Figure 1.1.b shows an interface integration system (indicated by a dotted box) which provides a global query interface to a set of airliner and travel agent sources. □

The integration of source query interfaces typically involves three major tasks [43, 92]: *schema extraction*, *schema matching*, and *schema merging*. In the rest of this section, we describe these tasks in detail. We will use examples in Figure 1.2 for the illustration.

1.3.1 Schema Extraction

The goal of schema extraction is to infer the schema of a query interface from its HTML representation. As discussed above, each source interface contains a set of query attributes. For example, Figure 1.2.a (left) shows a query interface Q_1 to an airfare source. There are 11 query attributes on Q_1. A query attribute may have a label, denoting what the attribute is about. For example, the first attribute on Q_1 has a label Depart City. A query attribute may also have a set of instances. For example, the last attribute Class on Q_1 has a set of instances: {Economy, First Class, Business}.

Furthermore, as we can observe from Q_1, related attributes are typically placed together on the query interface, forming a group. And closely related attribute groups may be further grouped into a supergroup. As

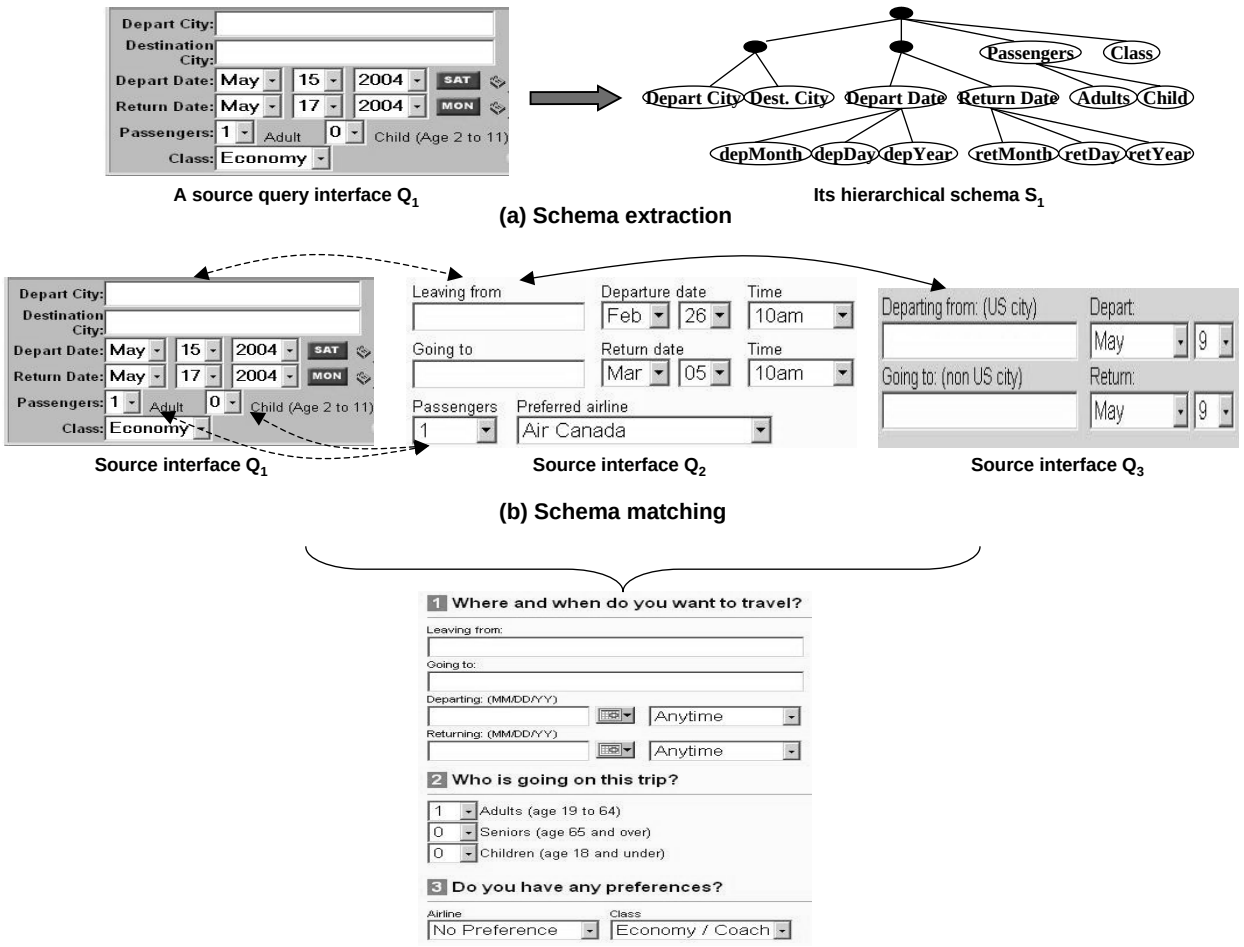


Figure 1.2: Three tasks in interface integration

a result, query interfaces are typically hierarchical. In addition, attributes and attribute groups are intuitively ordered. For example, attributes Adult and Child on Q_1 form a group with a group label Passengers. Note that attributes Adult and Child are intuitively ordered such that Adult appears before Child on the interface.

But the HTML form representation of a query interface only specifies the ways of rendering attributes (e.g., using text input field to display attribute Depart City on Q_1) and placing attributes and their labels on the interface. But the association of attributes with labels and the relationships among the attributes are **not** explicitly given on the query interface and need to be determined by a schema extraction algorithm. For example, given Q_1 as the input, such an algorithm should be able to obtain the schema of Q_1 as shown in Figure 1.2.a (right).

In the rest of the dissertation, we may use interface and schema (of source interface) interchangeably,

when it is clear from the context.

1.3.2 Schema Matching

The goal of schema matching is to accurately determine the semantic mappings among the attributes from different source interfaces. Typically, there are two types of mappings: simple and complex mappings. A simple mapping is a 1-1 semantic correspondence between two attributes. For example, consider interfaces Q_1, Q_2 and Q_3 shown in Figure 1.2.b. Attributes Depart City on Q_1, Leaving from on Q_2, and Departing from US city on Q_3 are all 1-1 corresponding to each other.

The correspondence among attributes may also be complex. In particular, we observe that 1-m mappings frequently occur among attributes [107]. A 1-m mapping refers to a mapping where an attribute on one interface semantically corresponds to multiple attributes on another interface. For example, attribute Passengers on interface Q_2 matches with attributes Adult and Child on interface Q_1.

Schema matching is a well-known challenging problem. It is also a fundamental problem in numerous applications including data integration, data warehousing, and personal information management. But despite many works, schema matching is still a very difficult problem. And the problem of matching source interfaces is particularly challenging largely due to the following factors: (1) *scale:* there are typically a large number of Deep Web sources in any domain of interest; (2) *diversity*: the query interface of these sources often vary greatly in their vocabulary and structure, due to the autonomous nature of the sources; and (3) *lack of evidence:* to make things worse, often there is little evidence on the semantics of attributes. For example, it is very difficult to discover the mapping between attribute Depart City on Q_1 and attribute Leaving from on Q_2, since their labels are very different and neither of them has any instances.

1.3.3 Schema Merging

The goal of schema merging is to unify source query interfaces into a *well-formed* global query interface, based on the attribute mappings discovered by schema matching. For example, Figure 1.2.c shows a global query interface which unifies source query interfaces Q_1, Q_2 and Q_3.

We say that a global query interface is well-formed if it has the following three properties.

(1) *Conciseness:* Semantically similar attributes from different source interfaces should appear as a single attribute on the global interface. For example, attributes Depart City on Q_1, Leaving from on Q_2,

and Departing from US city on Q_3 are similar to each other. So they are combined into one attribute Leaving from on the global interface.

(2) *Completeness:* Attribute unique to some source interface (i.e., it is not semantically similar to any attribute on any other interfaces) should also be retained on the global interface. For example, attribute Class is unique to Q_1 and attribute Preferred airline is unique to Q_2. They are both retained on the global interface.

(3) *User-friendliness:* A global interface can have as many as 30–50 attributes in some complex domain. So it is very important that closely related attributes are properly grouped and intuitively ordered on the interface. For example, at the top-level, the global interface in Figure 1.2.c organizes attributes in three categories: location/date, passengers, and preferences (in this order). In addition, attributes within each category are also properly ordered. For example, attribute Leaving from (for flight origin) is intuitively placed before attribute Going to (for destination).

Compared to schema matching, the problem of merging schemas has received relatively little attention. As discussed earlier, source interfaces are greatly diversified due to the autonomous nature of the sources. As a result, *structural conflicts*, as exemplified by the fact that different interfaces may contain a different set of attributes and may represent and organize the attributes in very different ways, are prevalent over the interfaces. For example, while interface Q_1 in Figure 1.2.b groups the attributes by location and date, interface Q_2 groups similar attributes based on departure and return.

The resolution of structural conflicts over a large number of interfaces thus poses serious challenges and calls for a novel scalable solution. Traditional approaches to schema merging [9] are either manual or relying on a set of resolution rules prescribed by domain experts. As a result, it is difficult for these approaches to scale up to a large number of diversified schemas.

1.4 Limitations of Existing Integration Solutions

While schema integration has been studied for decades [24, 25, 56, 59, 64, 70, 83, 92], the integration of source query interfaces poses many *unique* challenges, and thus has received great attention recently [41, 43]. Unfortunately, current solutions suffer from several fundamental limitations.

- *Non-hierarchical modeling:* All current solutions model query interfaces with *flat* schemas. For example, a flat schema for interface Q_1 in Figure 1.2.a is simply a set of attributes {Depart City, Destination City, ...}, with no particular ordering among the attributes. Nevertheless, as we observed earlier, attributes on an interface may be grouped and ordered based on their relative semantics, resulting in a much richer structure of the interface.

- *1:1 mapping assumption:* Most of the solutions consider only the 1:1 mappings among interface attributes. But, as we observed earlier, more complex mappings are pervasive over the interfaces. As a result, the mappings identified by these solutions may be incomplete or contain many incorrect ones. This will seriously impact the performance of schema merging which requires highly accurate attribute mappings.

- *Laborious parameter tuning:* Existing schema matching solutions typically have a large number of parameters to be tuned in order to yield good performance on a specific domain. Furthermore, when the system is applied to another domain, these parameters frequently need to be re-tuned. The tuning process is often done in a trial-and-error fashion, without any principled guidance.

- *Blackbox operations:* Almost all the existing solutions perform the integration in a blackbox-like fashion, where the integrator is typically only an observer once the integration process starts, and the whole process needs to be restarted from the scratch if anything goes wrong. As noted earlier, schema matching is a long-standing difficult task and it is very unlikely that an automatic solution could yield a perfect accuracy in any non-trivial domain. Therefore it becomes crucial for an integration system to be able to effectively utilize a minimal amount of human effort to greatly improve its performance.

- *Hampered by the lack of instances:* A unique challenge to matching interface schemas is the pervasive lack of attribute instances. In fact, as our experiments will show, in some domain there may be as high as 75% of the attributes which do not have any instances on their interfaces. It is particularly difficult to match these attributes, since we have to determine their semantics solely base on their labels. As we observed earlier, similar attributes may have very different labels. In addition, labels may be very ambiguous. For example, attribute with a label Job type on a job-search query interface could denote the *duration* of jobs such as permanent and temporary. But it could also denote the *specialty* of jobs such as accountant, clerk, and lawyer.

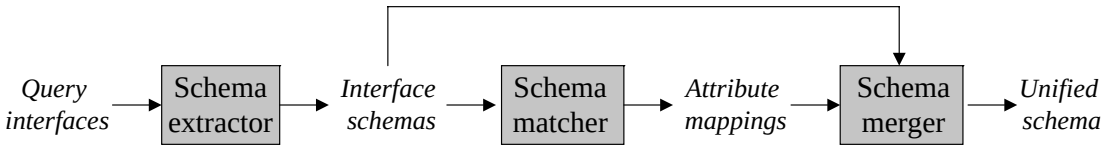


Figure 1.3: The IceQ architecture

- *Simplified merging solution:* Since all existing solutions assume that interfaces as flat, the global interfaces they produce simply contain a set of arbitrarily arranged attributes, with no particular grouping and ordering among the attributes. As a result, the interfaces produced by these solutions are not user-friendly and very difficult to understand and formulate queries.

1.5 Contributions of this Dissertation

To address these limitations, we have developed an interface integration system IceQ. IceQ provides an effective end-to-end solution to the interface integration problem.

Figure 1.3 shows the architecture of IceQ. It consists of three major components: schema extractor, schema matcher, and schema merger. IceQ accepts as input a set of source query interfaces represented in HTML form, and outputs a unified schema for the global query interface. It proceeds as follows. First, for each query interface, the *schema extractor* determines the attributes on the interface as well as the grouping and ordering relationships among the attributes. It outputs the schema of the interface represented as an ordered tree. Next, the *schema matcher* takes source interface schemas as the input, and discovers both 1:1 and complex mappings among the attributes over different interfaces. Finally, the *schema merger* constructs a unified schema by merging the source schemas based on the attribute mappings identified by the schema matcher.

In developing IceQ, we make the following contributions.

On interface modeling and schema extraction

- *Hierarchical modeling of query interfaces:* We show that hierarchical schema, such as ordered tree, can capture the semantics of the interface more precisely. Furthermore, we show that the structure of the interface can be effectively exploited to help identify the mappings of the attributes on the interface.

- *Automatic extraction of hierarchical schema:* We propose a *novel* spatial clustering-based algorithm to determine the structure of the interface based on its visual representation. Furthermore, we develop a *novel* label attachment algorithm which exploits several observations on the interface annotation process to accurately determine the label of *both* attributes and attribute groups on the interface.

On the problem of schema matching

- *Effective large-scale schema matching via clustering:* We develop a *novel* clustering-based schema matching algorithm and show that it can accurately identify mappings of attributes over varied real-world domains. The key idea is to view the problem of identifying mappings of attributes over a large number of schemas as a clustering problem. By matching a large number of schemas at once, the algorithm can achieve the so-called "bridging" effect to help match the attributes which have little in common themselves and therefore greatly improve the matching accuracy. The bridging effect is in spirit similar to the idea of reusing past identified mappings [83] and can be illustrated with the following example.

  Consider again the three interfaces shown in Figure 1.2.b. As we observed earlier, it may be very difficult to identify the mapping between attribute Depart City on Q_1 (denoted as A) and attribute Leaving from on Q_2 (denoted as B), given that their labels are very different and neither of them has any instances. Nevertheless, we observe that the label of A is similar to the label of attribute Departing from US city on Q_3 (denoted as C). Note that the two labels share two common words, depart and city, after stemming [89] (details will be given in Chapter 3). Further, the label of B is also similar to the label of C, since both labels have a common word "from". Note that "from" does *not* appear in the label of A. Note also that "from" is a content word in the airfare domain. Therefore, by considering matching attributes on these interfaces at the same time, attribute C may be effectively exploited to serve as a "bridge" to help relate attributes A and B.

- *Handling complex mappings:* We show that complex mappings, such as 1:m mappings, frequently occur among the attributes on the interfaces. We develop several approaches which exploit the characteristics of the attributes as well as the structure of the interfaces to discover complex mappings among the attributes from two interfaces. We further propose a novel three-staged approach which extends the clustering-based matching algorithm to effectively identify complex mappings among

the attributes over *all* the interfaces. The proposed approach achieves a "bridging" effect in finding complex mappings similar to that in finding 1:1 mappings as described above.

- *User interaction and active parameter learning:* We put human integrators back in the loop and present a novel approach which learns threshold parameters in the integration system by *selectively* asking the integrators certain questions. We further propose several approaches to determining the situations where the interaction between the system and the integrators is necessary to resolve the uncertain mappings, with the goal of reducing the amount of user interaction to the minimum. To the best of our knowledge, this is the *first work* on the active learning of parameters in a schema matching algorithm.

- *Exploiting external knowledge:* To address the problem of lacking instances, we develop WebIQ, a learning component for gathering instances of attributes from the Web, via novel adaptations of the question-answering techniques commonly employed in Artificial Intelligence. The gathered instances are then utilized to augment the interface schemas to assist in matching similar attributes.

  The key idea is to gather attribute instances from the Web by posing extraction queries to search engines and obtain instance candidates from the search results. For example, consider gathering instances for an attribute with label Departure city. One of the extraction queries for this attribute will be "departure cities such as", which is then sent to Google to obtain a list of result snippets. Finally, instance candidates are obtained from the snippets by extracting the noun phrases immediately following "such as". For example, from the snippet "... departure cities such as Chicago, Boston, and New York ...", the approach will obtain three instance candidates: Chicago, Boston, and New York.

On the problem of schema merging

- We present a first systematic study on the problem of merging a *large number* of interface schemas. We propose a novel *optimization* framework for schema merging, where each interface schema in essence expresses certain constraints on the unified schema and the goal is to construct a unified schema such that these constraints are maximally satisfied. Unfortunately, the optimization problem can be shown to be a NP-complete problem.

- To address this challenge, we develop an approximation algorithm LMax based on the idea of *clus-*

tering aggregation [33]. We further propose several novel metrics to objectively measure the quality of constructed unified schemas. Extensive experiments have been conducted over varied real-world domains and results indicate that the proposed solution can consistently produce well-formed unified schemas.

1.6 Outline

The rest of the dissertation is organized as follows. Chapter 2 proposes hierarchical modeling of query interfaces and presents the schema extraction algorithm for obtaining the hierarchical schema of a query interface from its HTML representation. Chapter 3 describes the proposed interactive clustering-based schema matching algorithm. Chapter 4 presents the Web-based instance learning component WebIQ. Chapter 5 proposes the optimization framework for schema merging and describes the merging algorithm LMax. Chapter 6 describes related works and Chapter 7 discusses future directions and concludes the dissertation.

Parts of this dissertation are based on several publications in conferences and journals. In particular, the schema matching algorithm is based on the SIGMOD-04 paper [107]. The description of WebIQ is based on the ICDE-06 paper [104]. LMax is described in the ICDM-05 paper [103]. The proposed schema extraction algorithm is based on a recent submission to TODS [106]. Finally, discussions on result merging in future directions are based on the VLDB-TES workshop paper [105].

Chapter 2

Schema Extraction

In this chapter, we first describe query interfaces, and show how prior work has modeled such interface with a flat set of attributes and how we model it with a tree of attributes (Section 2.1). We then describe how to extract such tree from the HTML page that contains the query interface (Section 2.2) and how to find labels on the interface for the nodes in the tree (Section 2.3). Finally, we present our experimental results (Section 2.4) and summarize the chapter (Section 2.5).

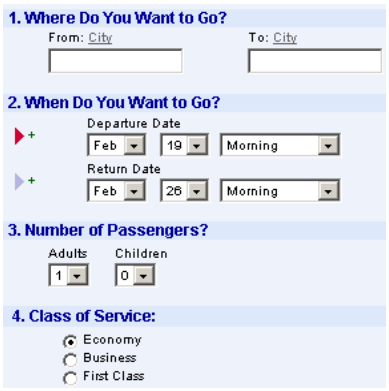
2.1 Flat vs. Hierarchical Modeling of Query Interfaces

Query interfaces are typically written in HTML forms. For example, Figure 2.1.a shows a query interface Q in the airfare domain and Figure 2.1.b shows the HTML form script of Q.

A query interface can be modeled using multiple attributes. For example, Q contains 11 attributes whose details are shown in Figure 2.1.c. Note that the attributes are numbered in the order of their appearance (left-right, top-down) on the interface. Each attribute consists of three components: label, (internal) name, and domain.

- **Label:** The label of an attribute is a piece of text on the query interface, which denotes the meaning of the attribute to the user. For example, the first attribute on Q (i.e., f_1) has a label From: City.
- **Name:** The name of an attribute is the internal name of the attribute given in the HTML script for the identification purpose. For example, the name of attribute f_1 is origin.
- **Domain:** The domain of an attribute is a set of values the attribute may take. For example, the domain of the attribute f_9 (with label Adults) on Q is $\{1,2,...,6\}$.

Note that an attribute may be represented in a variety of ways on the query interface: (1) an *input field* (e.g., attribute f_1 on Q), where the user may enter any suitable value; (2) a *selection list* (e.g., attribute f_3),



(a) An airfare query interface Q

```
...
<TABLE cellSpacing=0 cellPadding=0 width=638 border=0>
  <TBODY>
  <TR bgColor=#dde9fb>
    <TD width=608 colSpan=2>
      <FONT class=moduleHeader2>
        1. Where Do You Want to Go?</FONT></TD></TR>
  <TR>
    <TD width=578>
      <TABLE cellSpacing=0 cellPadding=0 border=0>
        <TBODY>
        <TR> <TD>From:</TD> <TD>To:</TD> </TR>
        <TR>
          <TD><INPUT maxLength=30 size=10 name=origin
               maxlenth="30"></TD>
          <TD class=moduleText><INPUT size=10
               name=destination maxlenth="30"> </TD>
        </TR>
        </TBODY>
      </TABLE>
    </TD>
  </TR>
  </TBODY>
</TABLE>
...
```

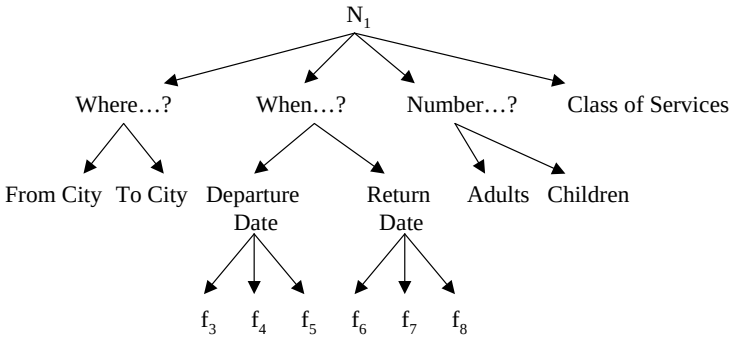
(b) The HTML script of Q

Attribute	Name	Label	Domain
f_1	origin	From: City	{s|s is any string}
f_2	destination	To: City	{s|s is any string}
f_3	departureMonth	""	{Jan, Feb, …, Dec}
f_4	departureDay	""	{1, 2, …, 31}
f_5	departureTime	""	{1am, …, 12pm}
f_6	returnMonth	""	{Jan, Feb, …, Dec}
f_7	returnDay	""	{1, 2, …, 31}
f_8	returnTime	""	{1am, …, 12pm}
f_9	numAdultPassengers	Adults	{1, 2, …, 6}
f_{10}	numChildPassengers	Children	{0, 1, …, 5}
f_{11}	cabinClass	Class of Services	{Economy, …, Business}

(c) The attributes on Q

{from city, to city, …, class of services}

(d) A flat schema of Q



(e) A hierarchical schema of Q

Figure 2.1: A query interface, its HTML script, attributes, and schemas

where the user may only select from a list of pre-defined choices; (3) a *radio-button group* (e.g., attribute f_{11}), where each button in the group provides an exclusive choice, and the domain of the attribute is the set of all choices while the name of the attribute is taken to be the name of the radio button group; and (4) a *checkbox group*, which is similar to a group of radio buttons except that here the user may select more than

one choices at a time.

Note also that label is visible to the user while name is not. As a consequence, words in the label are usually ordinary words which can be understood semantically, while words in the name are often concatenated or abbreviated. Nevertheless, we found that the name of an attribute often can be very informative, and particularly useful when the attribute does not have a label.

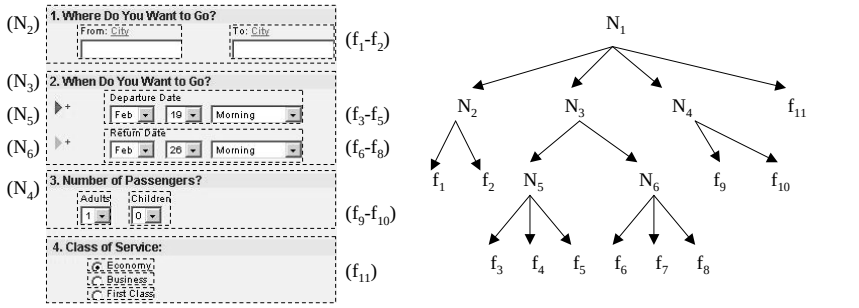
Current works represent a query interface with a flat set of attributes, as defined above. For example, Figure 2.1.d shows such flat schema of Q. But in fact, as Q shows, closely related attributes (e.g., f_1 and f_2, both on the location of the flight) may be grouped together. Furthermore, attributes and attribute groups may be intuitively ordered (e.g., f_1, for origin, is placed before f_2, for destination). As a result, the query interface has a much richer structure. Such a structure conveys domain knowledge and may be exploited for the effective integration of interfaces.

To capture both the grouping and ordering relationships of attributes on a query interface, we model query interface with a hierarchical schema. Figure 2.1.e shows an example of such hierarchical modeling, which is technically an ordered tree. A leaf node in the tree corresponds to an attribute on the interface. An internal node corresponds to a group or a super-group of attributes on the interface. Nodes with the same parent are sibling nodes. Sibling nodes are ordered by the sequence of their corresponding attributes or attribute groups (if they are internal nodes) appearing on the interface.

Note that nodes are annotated with the labels of their corresponding attributes or attribute groups. If a node does not have a label, its ID is shown instead, where N_i's represent internal nodes and f_j's leaf nodes. In the rest of the paper, we may also call the nodes of the tree as the elements of the schema.

Such a hierarchical modeling of query interfaces greatly improves the performance of both schema matcher and schema merger:

- We show that the ordering relationships of the attributes may be exploited to resolve the ambiguous 1-1 matches (Section 3.2.2);
- We show that the grouping relationships of the attributes may be exploited to effectively identify the attributes involved in complex matches (Section 3.2.3);
- We show that both the grouping and ordering relationships of the attributes are critical for the schema merger to construct a user-friendly global query interface [103].



(a) Query interface Q (b) S_Q, the schema of Q (*before* label attachment)

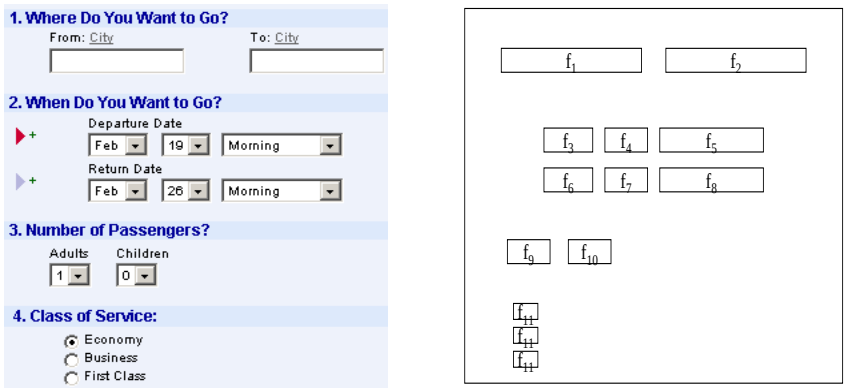
Figure 2.2: Example of extracting tree structure of an interface

From now on, when we refer to such modeling, we use the phrase "query interface". Extracting such query interfaces is difficult for the following reasons. First, we must group the attributes appropriately. Next, we must extract the labels and assign them to the right places. We describe how to extract such interfaces next. Note that the names of attributes can be easily obtained from the HTML script of the query interface. If an attribute is represented as a selection list, then its values can also be easily obtained from the option sub-elements of the list. We will describe in Section 2.3 how to obtain the values of an attribute represented as a radio-button or checkbox group.

2.2 Extracting the Tree Structure of an Interface

In this section, we describe a structure extraction algorithm based on spatial clustering. The algorithm takes as the input a query interface (e.g., Q in Figure 2.2.a) and produces a unannotated ordered-tree schema of the interface (e.g., S_Q in Figure 2.2.b). In the next section, we will describe a label attachment algorithm which then assign the labels from the interface to the nodes in the schema to produce the final schema (e.g., Figure 2.1.e).

The main idea of the algorithm is to exploit the spatial relationships (e.g., proximity, alignment, and direction) of attributes on the query interface to effectively discover both the grouping and ordering relationships among the attributes. In the following, we start by describing a basic version of the algorithm which produces a schema tree where each node can have at most two children. We then describe how to remove this limitation via n-way clustering. Next, we discuss how to exploit other information such as lines separators to help determine the grouping relationships of attributes. Finally, we present the complete extraction algorithm.



(a) A query interface Q (b) Q's attribute blocks

Figure 2.3: A query interface and its attribute blocks

2.2.1 Structure Extraction via Spatial Clustering

The basic version of the extraction algorithm can be regarded as a conventional hierarchical agglomerative clustering algorithm [49] where the objects to be clustered are *attribute blocks*. Attribute block is the spatial representation of an attribute, which can be obtained as follows.

If an attribute f is rendered as an input field (e.g., f_1 and f_2 on interface Q in Figure 2.3.a) or a selection list (e.g., $f_3 - f_{10}$ on Q), then f's attribute block is taken to be the smallest rectangular region enclosing the input field or the selection list (see Figure 2.3.b). On the other hand, if f is represented as a group of radio buttons (e.g., f_{11}) or checkboxes, then f's attribute block is taken to be the smallest rectangular region enclosing all the radio buttons or checkboxes in the group (see Figure 2.3.b).

In the following, we may also denote an attribute block B as $[(x, y), (m_x, m_y)]$, where (x, y) is the top-left corner of B (with x as the x-coordinate and y as the y-coordinate), and (m_x, m_y) is the bottom-right corner of B.

We consider three types of spatial relations between the blocks: *topological relations* (contain, overlap, and disjoint), *direction relations* (above, below, left, and right), and *alignment relations* (top/bottom-aligned and left/right-aligned).

Definition 1 (Topological Relations) A block U is *contained* in a block V if $\forall p \in U$ (i.e., p is a point in U), we have $p \in V$. A block U *overlaps* with a block V if $\exists p \in U$ such that $p \in V$ and $\exists q \in U$ such that $q \notin V$. A block U is *disjoint* with a block V if if $\forall p \in U$, we have $p \notin V$. □

Definition 2 (Direction Relations) A block U is *above* (*below*) a block V if $\forall p \in U$ and $\forall q \in V$, we have $p_y > q_y$ ($p_y < q_y$), where p_y denotes p's y-coordinate. A block U is to the *left* (*right*) of a block V if $\forall p \in U$ and $\forall q \in V$, we have $p_x < q_x$ ($p_x > q_x$), where p_x denotes p's x-coordinate. □

Definition 3 (Alignment Relations) Consider two blocks U = [(x,y), (m_x, m_y)] and V = [(s,t), (m_s, m_t)]. U is *left-aligned* (*right-aligned*) with V if $x = s$ ($m_x = m_s$); and U is *top-aligned* (*bottom-aligned*) with V if $y = t$ ($m_y = m_t$). □

Distance Function: Intuitively, if two blocks are close to each other and aligned, it is likely that they belong to the same group. Accordingly, we define a distance function between two blocks U and V, denoted as dist(U, V), as follows:

$$\mathrm{dist}(U, V) = \frac{\text{point-dist}(U, V)}{\mathrm{align}(U, V)}. \tag{2.1}$$

point-dist(U, V) is the minimum Euclidean distance between any two points in U and V. align(U, V) is given by left-align(U, V) + right-align(U, V) + top-align(U, V) + 2 $*$ bottom-align(U, V), where left-align(U, V) takes the value of one if U is left-aligned with V, and zero otherwise. Other alignment functions are similarly defined. If U and V are not aligned, align(U, V) is set to one, i.e., no adjustment will be made to the point-dist. Note that the weight coefficient for bottom-align is set to two since intuitively two adjacent blocks on the same line are more likely to be closely related.

Based on the above block distance function, the distance between two clusters of attribute blocks can be defined as follows. Consider a cluster C which contains a set of attribute blocks $S = \{B_1, B_2, ..., B_k\}$. We define a block for the cluster C, denoted as B_C, as the *smallest* rectangular region enclosing all the attribute blocks in S. Then, the distance between two clusters C and C' is measured by dist(B_C, $B_{C'}$). We are now ready to describe the clustering algorithm.

Clustering: The algorithm accepts as input a set of attributes on a query interface, where each attribute is represented by its corresponding attribute block as described above; and outputs a hierarchical clustering over the attributes. It starts by putting each attribute in a cluster by itself, and then repeatedly merges two clusters with the minimum distance, until all the attributes are put into a single cluster.

Note that the algorithm produces only *binary* clusterings, i.e., a cluster can only have two sub-clusters. This does not correspond well to the grouping relationships of attributes, since an attribute group may

contain more than two sub-groups of attributes. For example, attribute group $\{f_3, f_4, f_5\}$ on the interface Q (Figure 2.1.a) contains three attributes. To cope with this, we extend the algorithm to handle n-way clustering.

2.2.2 N-way Clustering

The extended algorithm works similarly as the basic one: initially we have a set of clusters, each containing a single attribute, and we repeatedly merge the clusters until we have a single cluster with all the attributes. The key difference is in the merge operation: rather than immediately merging two clusters with the minimum distance, it first expands them into a proximity set of clusters, and then merges all the clusters in the proximity set in a single step.

Specifically, consider two clusters C_1 and C_2, where dist(B_{C_1}, B_{C_2}) = d. A proximity set with respect to C_1 and C_2, denoted as S, can be obtained as follows. To start with, we set $S = \{C_1, C_2\}$. We then use d as the reference distance, and keep growing S by adding a new cluster C_x such that $\exists C_i \in S$, $|\text{dist}(B_{C_x}, B_{C_i}) - d| < \delta * d$, where δ is a small constant (e.g., $\delta = .1$ in our experiment). This growing process stops when no such cluster can be found.

Example 3 Suppose clusters C_1 = $\{f_3\}$, C_2 = $\{f_4\}$, and C_3 = $\{f_5\}$. Then $\{C_1, C_2, C_3\}$ may be a proximity set with respect to C_1 and C_2, since the distance between C_3 and C_2 is very close to the distance between C_1 and C_2. □

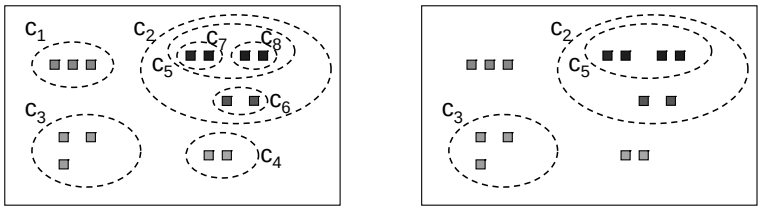
2.2.3 Exploiting Non-Distance Information as Constraints

Besides the distance among the attributes, query interfaces may also contain other information such as section titles or horizontal lines, which can be exploited to help determine the grouping relationships of the attributes. For example, the attributes on the interface Q (Figure 2.1.a) can also be divided into four sections by the section titles: “1. Where Do You Want to Go?”, “2. When Do You Want to Go?”, so on.

In this section, we describe how to search for these additional information on the query interface and how to exploit them to obtain a partial clustering over the attributes. We will describe in Section 2.2.4 how to incorporate the obtained partial clustering to constrain the merging process in the spatial clustering algorithm.

Pattern Types	*Examples*
Separator-based	a. Fields separated by a set of section labels which are left-aligned and have the same large font. b. Fields separated by a set of left-aligned horizontal lines.
Alignment-based	c. Multiple rows of fields which are top and bottom-aligned along the row, and left and right-aligned across the rows.
Indentation-based	d. A group of fields which are all indented relative to a label which is located right above and has a large font.
HTML script-based	e. A group of radio buttons with the same group name given in the script.

Table 2.1: Grouping patterns



(a) A complete clustering (b) A partial clustering

Figure 2.4: Partial vs. complete clusterings

Grouping Patterns: To systematically search for these information, we employ a set of grouping patterns. Each grouping pattern specifies a way of grouping some attributes on the interface. These grouping patterns fall into three categories (see Table 2.1 for examples on each category).

- *Separator-based* patterns, which utilize separators such as section titles and horizontal lines to divide the attributes into groups. Note that the labels which have a larger font (compared to the most common font among the labels) and are located at the left-most of the interface are regarded as section titles.
- *Alignment-based* patterns, which identify groups of attributes which are highly aligned to one another.
- *Indentation-based* patterns, which identify groups of attributes based on their indentation relative to a label.

Partial Clustering: The above patterns may then be employed to obtain a partial clustering over the attributes on the query interface. Note that such a partial clustering may not be a complete clustering, rather it gives a rough idea of how the final complete clustering should look like. For example, the partial

clustering might not indicate the grouping relationships of the attributes within each section on the interface Q in Figure 2.1.a. Partial clusterings can be formally defined as follows.

Definition 4 (Partial Clustering) Consider a set of attributes $S = \{f_1, f_2, ..., f_n\}$. A *flat* partial clustering $\mathcal{P}$ over the attributes in S is a set of subsets of attributes, i.e., $\mathcal{P} = \{S_1, ..., S_k\}$, such that $S_i \subset S$ and $S_i \cap S_j = \emptyset$ for $i \neq j$. Note that $\mathcal{P}$ might not have the property that $\cup_{1 \leq i \leq k} S_i = S$. Otherwise, $\mathcal{P}$ is a *complete* clustering over the attributes in S.

Such a partial clustering may be further formed over *some* of the subsets in $\mathcal{P}$. Proceed recursively, the resulted nested clustering is called a *hierarchical* partial clustering over the attributes in S. □

Since we are only concerned with hierarchical clusterings over the attributes, we will simply use partial clusterings to refer to hierarchical partial clusterings.

Example 4 Figure 2.4 shows a partial clustering vs. a complete clustering over the same set of attributes, where clusters are represented by dotted ovals. We observe that at the first level, the complete clustering forms four clusters over the attributes, but only two of them (C_2 and C_3) are given in the partial clustering. □

Obtain Partial Clustering: Based on the above discussions, we are now ready to describe PRECLUSTER, a procedure which takes as input a set S of attributes on a query interface Q, and outputs a partial clustering P over the attributes in S. PRECLUSTER proceeds in a top-down fashion. It first finds attributes groups among the attributes in S by applying a set G of grouping patterns. These attribute groups form the top level clusters of the partial clustering. It then recursively finds sub-groups among the attributes within each group.

Specifically, PRECLUSTER consists of the following steps. (a) *Pattern matching:* apply the patterns in G on S. Each pattern returns a set of subsets of attributes in S, denoted as $\{S_1, ..., S_k\}$, where $S_i \subset S$ and $S_i \cap S_j = \emptyset$ for $i \neq j$. Let G_S be a set of all such subsets given by the patterns in G. If $G_S = \emptyset$, then stop. (b) *Maximization:* from the subsets in G_S, select a set of maximum subsets, denoted as G'_S. A subset $S_x \in G_S$ is a maximum subset if there does not exists $S_y \in G_S$ such that $S_y \subset S_x$. (c) *Recursion:* if there is at least one subset in G'_S which has more than two attributes, recursively apply steps a–b on each such subset in G'_S.

EXTR(S) $\rightarrow \mathcal{T}$:
Input: S, a set of attributes on an interface
Output: $\mathcal{T}$, an unannotated ordered tree schema

1. Utilize grouping patterns to obtain partial clustering:
   $\mathcal{P} \leftarrow$ PRECLUSTER(S)
2. Form initial clustering:
   /* $\mathcal{C}$ contains a singleton cluster for each attribute $f \in S$ */
   $\mathcal{C} \leftarrow \{\{f\} \mid f \in S\}$
3. Repeat the following steps until all attributes are in one cluster:
   /* each iteration performs a n-way constrained merging operation */
   a. Obtain clusters to be considered in the current iteration:
      $\mathcal{C}_{\mathcal{P}} \leftarrow$ CONSTRAIN($\mathcal{C}$, $\mathcal{P}$)
   b. Find two clusters C_1, $C_2 \in \mathcal{C}_{\mathcal{P}}$ with the minimum distance
   c. Expand them into a proximity set:
      $X \leftarrow$ OBTAINPROXIMITYSET(C_1, C_2, $\mathcal{C}_{\mathcal{P}}$)
   d. Merge clusters in X into a new cluster C_X
   e. Evaluate distances of C_X with remaining clusters via Formula 2.1
4. $\mathcal{H} \leftarrow$ the hierarchical clustering output by step 3
5. Order attributes and attribute groups in $\mathcal{H}$:
   $\mathcal{T} \leftarrow$ ORDER($\mathcal{H}$)
6. Return $\mathcal{T}$

Figure 2.5: The structure extraction algorithm

The maximum subsets obtained over the iterations of the above recursive procedure form a top-down partial clustering over the attributes on the interface Q.

So given a partial clustering (e.g., Figure 2.4.a) over the attributes on a query interface, the goal of the spatial clustering algorithm is in a sense to obtain a complete clustering (e.g., Figure 2.4.b) which respects the partial clustering. As we will show next, one way of doing this is to use the partial clustering to constrain the merging process of the algorithm.

2.2.4 The Structure Extraction Algorithm

Figure 2.5 shows the complete structure extraction algorithm **EXTR**. **EXTR** accepts as input S, a set of attributes on an interface, and outputs $\mathcal{T}$, an unannotated ordered-tree schema of the interface. At the high level, **EXTR** is a hierarchical agglomerative n-way clustering algorithm where the merging process is constrained so that it does not violate the partial clustering obtained by PRECLUSTER.

It proceeds as follows. First, it applies PRECLUSTER to obtain a partial clustering $\mathcal{P}$ over the attributes in S. $\mathcal{P}$ is then used to constrain the merging process via the CONSTRAIN function at step 3(a). Given the current clusters in C and the partial clustering $\mathcal{P}$, CONSTRAIN finds a minimum cluster $C_m \in \mathcal{P}$ such that

C_m contains a set of clusters in C, denoted as $C_{\mathcal{P}}$. Note that a cluster $C_m \in \mathcal{P}$ is minimum if $\nexists C'_m \in \mathcal{P}$, such that C'_m also contains all clusters in $C_{\mathcal{P}}$ and $C'_m \subset C_m$. If such a minimum cluster $C_m \in \mathcal{P}$ is found, CONSTRAIN returns the corresponding $C_{\mathcal{P}}$ as the output; otherwise, it returns C as $C_{\mathcal{P}}$.

Example 5 Suppose the partial clustering $\mathcal{P}$ is as given in Figure 2.4.b. Then in the first iteration of the step 4, $C_m = C_5$ and $C_{\mathcal{P}}$ is a set of singleton clusters with the attributes in C_m. □

Then, in the remaining of step 3, only the clusters in $C_{\mathcal{P}}$ are considered. First, two clusters $C_1, C_2 \in C_{\mathcal{P}}$ with the minimum distance are chosen. C_1 and C_2 are then expanded into a proximity set X as described in Section 2.2.2. Note that X only contains the clusters in $C_{\mathcal{P}}$. Next, the clusters in X are merged into a new cluster C_X. Finally, the distances of C_X with the remaining clusters are evaluated, before the next iteration.

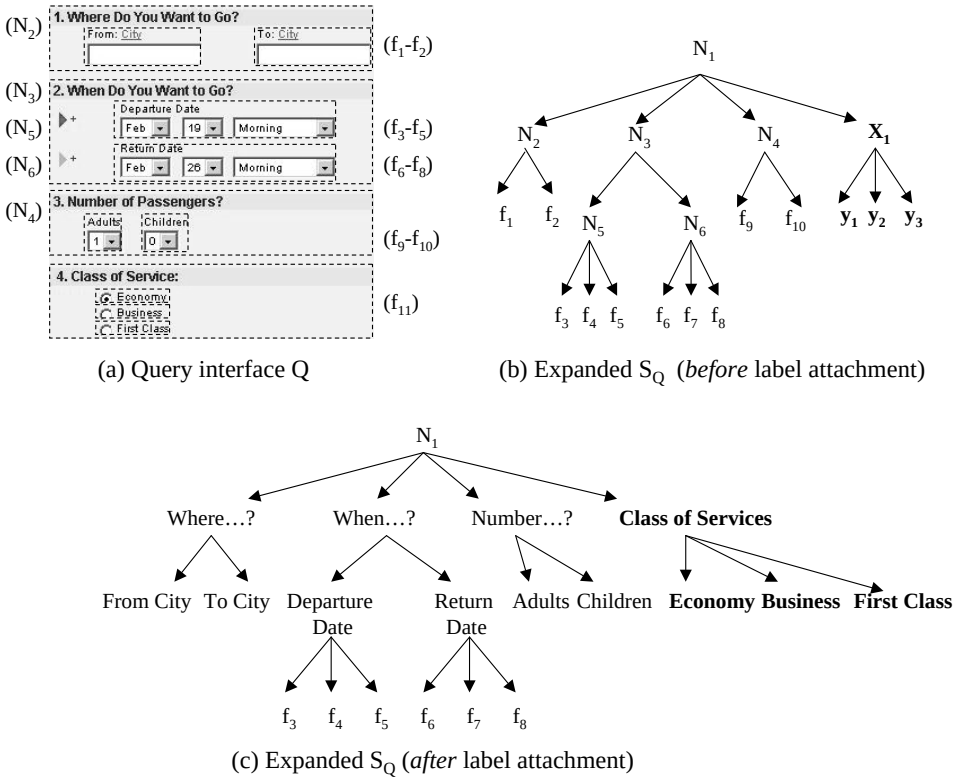
The result of step 3 is a hierarchical clustering $\mathcal{H}$ over the attributes on the interface. $\mathcal{H}$ corresponds to an *unordered* schema tree of the interface. Finally, step 5 orders the nodes in $\mathcal{H}$ to produce an ordered schema tree $\mathcal{T}$ via the ORDER function.

ORDER considers the internal nodes of $\mathcal{H}$ in turn, and for each internal node I, it arranges I's child nodes by the spatial location of their corresponding attributes or attribute groups on the interface. Specifically, suppose I has k children I_1, I_2, ..., I_k. Denote the smallest rectangular box which encloses all the attributes (i.e., leaf nodes) of the subtree rooted at I_i as B_{I_i}. Then, I_i precedes I_j in the ordering, if one of the following two conditions holds: (1) B_{I_i} and B_{I_j} overlaps in the y-direction, and B_{I_i} is to the *left* of B_{I_j}; or (2) B_{I_i} and B_{I_j} does not overlap in the y-direction, and B_{I_i} is above B_{I_j}. Such an ordering corresponds to the intuitive left-right top-down viewing sequence of the attributes on the interface by the users.

2.3 Extracting and Attaching the Labels

In Section 2.2.4, we described a structure extraction algorithm which takes as the input a query interface (e.g., Q in Figure 2.6.a) and produces a unannotated schema of the interface (e.g., S_Q in Figure 2.6.b). In this section, we describe a label attachment algorithm which finds the labels from the interface for the nodes in the schema.

As described earlier, if an attribute is represented as a group of radio buttons or checkboxes, then its values are the labels of the individual radio buttons or checkboxes. In order to also extract these labels, we expand the schema of the interface before label attachment so that every such attribute (e.g., f_{11} on Q) is



(a) Query interface Q

(b) Expanded S_Q (*before* label attachment)

(c) Expanded S_Q (*after* label attachment)

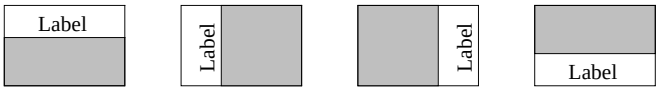
Figure 2.6: Example of label attachment

transformed into an attribute group (e.g., X_1 in Figure 2.6.b) which contains as many (pseudo) attributes as the number of radio buttons or checkboxes for the original attribute (e.g., y_1, y_2, and y_3). Then, after the label attachment is finished, the pseudo attributes will be removed from the expanded schema (e.g., Figure 2.6.c) to produce the final schema (e.g., Figure 2.1.e). Note that the labels for the pseudo attributes (e.g. Economy, Business, First Class) will become the values of the original attribute and the label of the attribute group (e.g., Class of Service) will become the label of the original attribute.

While there have been some works on label attachment [82], they either assume that query interfaces are flat and thus do not consider the attachment of group labels, or only handle groups of radio buttons and checkboxes (see related work section for more details). Furthermore, the current solutions commonly employ distance-based heuristics where labels are attached to the attributes with the smallest distances. Such heuristics may not work well, especially for group labels. For example, consider the interface snippet in Figure 2.7, which contains a group of two attributes (one for each selection list). We observe that the group label Passengers is closer to the first attribute than its actual label Adult.

To address these challenges, we take a closer look at the process of annotating attributes and attribute

Figure 2.7: Examples of label attachment where distance-based methods fail



(a) Above (b) Left (c) Right (d) Below

Figure 2.8: Positions of the annotating label in an annotation block

groups on a query interface with labels. For each annotation, we define an *annotation block* as the smallest rectangular region enclosing the annotating label and the attribute or attribute group the label annotates. For example, Figure 2.6.a shows the annotation blocks (represented by dashed rectangular boxes) for the attributes and attribute groups on the interface Q. The following observations can be made.

First, *non-overlapping annotation blocks:* It is unusual that annotation blocks would overlap with each other. In other words, for any two annotation blocks, there may only be two possibilities: either they are disjoint or one is contained with another. This observation can be verified using Figure 2.6.a. For example, consider the annotation block B_{N_1} for the group N_1, which encloses the group label (i.e., 1. Where Do You Want to Go?) and the attributes in the group (i.e., f_1 and f_2). We can observe that B_{N_1} contains the annotation blocks for attributes f_1 and f_2 and does not overlap with any other annotation blocks.

Second, *label positioning:* A group label is usually located either *above* or to the *left* of the group, while an attribute label may also be located to the right of the attribute, but seldom located *below* the attribute. For example, all the group labels are located above the groups on the interface Q in Figure 2.6.a, and none of the attribute labels is located below the attributes. Figure 2.8 shows possible layouts of an annotation block and the respective positions of the annotating label.

2.3.1 The Label Attachment Algorithm

Motivated by the above observations, we propose a label attachment algorithm ATTACH. ATTACH accepts as input an unannotated schema tree $\mathcal{T}$ for an interface, and a set $\mathcal{L}$ of all labels on the interface. It annotates the nodes in $\mathcal{T}$ with the labels in $\mathcal{L}$, and returns an annotated schema tree $\mathcal{T}^a$. The main ideas of ATTACH are as follows.

ATTACH($\mathcal{T}$, $\mathcal{L}$) $\rightarrow$ $\mathcal{T}^a$:
Input: $\mathcal{T}$, an *unannotated* schema tree; $\mathcal{L}$, all labels on the interface
Output: $\mathcal{T}^a$, an *annotated* schema tree

1. Obtain a list of internal nodes of $\mathcal{T}$ in post-order:
   $<N_1, N_2, ..., N_k> \leftarrow$ POSTORDER($\mathcal{T}$)
2. Initialize L_a to contain all labels on the interface:
   a. Let L_a be a set of available labels
   b. $L_a \leftarrow \mathcal{L}$
3. For each N_i, annotate its child nodes via ATTACHONE:
   a. Let L_r be a set of unassigned labels
   b. for i = 1 to k do
      $L_r \leftarrow$ ATTACHONE(N_i, L_a)
      $L_a \leftarrow L_r$
4. Return $\mathcal{T}^a$ = the annotated $\mathcal{T}$

ATTACHONE(N, L_a) $\rightarrow$ L_r:
Input: N, an internal node; L_a, available labels
Output: L_r, remaining labels

1. Generate label candidates:
   a. Let $\mathcal{N}$ be a set of child nodes of N
   b. Let M be a $|\mathcal{N}| \times |L_a|$ matrix
   c. $M \leftarrow$ OBTAINATTACHMENTMATRIX($\mathcal{N}$, L_a)
2. Prune candidates based on annotation rules:
   a. Let M' be the updated attachment matrix
   b. $M' \leftarrow$ PRUNECANDS(M)
3. Assign labels to nodes based on M' via best-first strategy
4. Return L_r = unassigned labels in L_a

Figure 2.9: The label attachment algorithm

- *Bottom-up:* One way of annotating the nodes in a schema tree is to proceed in a bottom-up fashion: we start with the leaf nodes and annotate a node only when all of its child nodes have been annotated. For example, consider interface Q in Figure 2.6.a. We first find labels for attributes f_1 and f_2 before finding label for group N_1.

- *Group-based:* Rather than annotating nodes in isolation, we may consider the annotation of a node and its sibling nodes (i.e., nodes within the same group) together. Intuitively, knowing that a label is unlikely to be assigned to neighbor nodes helps determine the node which the label should be attached to.

Based on the above ideas, ATTACH considers the groups (i.e., internal nodes) in the schema tree in the *post-order*. For example, the groups in the schema S_Q (Figure 2.6.b) are considered in this order: N_2, N_5, N_6, N_3, N_4, X_1, N_1. For each group N, it annotates the child nodes of N via ATTACHONE described below.

For example, when $N = N_2$, ATTACHONE annotates attributes f_1 and f_2, and when $N = N_1$, ATTACHONE annotates N_2, N_3, N_4, and X_1. We now first define several necessary concepts.

Definition 5 (Attribute Set and Block of a Node) For each node x in a schema tree, we define an attribute set, denoted as $\mathcal{A}_x$, as a set of attributes (i.e., leaf nodes) in the sub-tree rooted at x; and a block, denoted as B_x, as the smallest rectangular region enclosing all the attributes in $\mathcal{A}_x$. □

ATTACHONE: ATTACHONE accepts as the input a group N and a set L_a of available labels. It assigns some labels from L_a to the child nodes of N and returns the unassigned labels. It proceeds in three major steps: candidate generation, candidate pruning, and match selection. We now describe them in detail.

(1) Candidate generation: For each child node x of N, ATTACHONE determines which labels in L_a may be assigned to x, according to the *non-overlapping annotation areas* observation. Specifically, a label l is regarded as a candidate label for x if the annotation block enclosing the label l and the attributes in $\mathcal{A}_x$, the attribute set of x, does not overlap with any attributes not in $\mathcal{A}_x$ and any other labels in L_a.

For example, label l = From: City (Figure 2.6.a) is a candidate label for f_1 since the annotation block enclosing l and f_1 does not overlap with any other attributes or labels. On the other hand, l may not be assigned to f_2, since the annotation block enclosing l and f_2 overlaps with attribute f_1 (and also another label To: City).

This step results in an attachment matrix M, whose rows correspond to the child nodes of N, and columns correspond to the labels in L_a. The entry $M[i, j]$ is one if the j-th label is a candidate label for the i-th child node of N, and zero otherwise.

(2) Candidate pruning: This step prunes the candidates in M according to the *label positioning* observation as well as the distances between labels and blocks. The pruned matrix is denoted as M'.

It proceeds as follows. First, all candidate labels for a node x are pruned if the distance between the labels and the node block B_x is larger than a threshold d. Next, if x is an attribute and has a candidate label which is *not* located *below* B_x, then all the labels below B_x are pruned. Finally, if x is an attribute group, then all its candidate labels which are located below or to the right of B_x are pruned.

For example, since attribute f_1 has a candidate label From: City located above it, another candidate label 2. When Do You Want to Go?, located below it, is pruned.

(3) Match selection: Based on the pruned attachment matrix M' from step 2, this step assigns labels to blocks via a ***best-first*** strategy, starting with the most confident assignments. Specifically, the following cases are considered in turn: (a) a label l can only be assigned to a node x and x does not have any candidates other than l; (b) a label l can only be assigned to a node x and l is inside B_N (i.e., the node block of N); (c) a label l can only be assigned to a node x and l is to the right of B_N; and (d) a label l can only be assigned to a node x, but not in case (b) or case (c). Note that cases (b) and (c) are considered before case (d), since the labels which are inside or to the right of B_N are unlikely to be a label for the group N.

For each case, all the entries in M' are checked. If an entry $M'[i, j]$ falls into the case, then the j-th label will be assigned to the i-th block and all entries at the i-th row and the j-column of M' will be set to zero. The above process is then repeated until none of the entries in M' falls into any of the cases.

For example, it can be shown that, after the candidate pruning step, the label FROM: CITY will be the only candidate label for the attribute f_1 and f_1 does not have any other candidate labels. Thus FROM: CITY will be assigned to f_1 according to case (a). For another example, consider assigning labels to a group of two attributes (g_1 and g_2, each represented by a selection list) on the interface snippet shown in Figure 2.7. First, Child (Age 2 to 11) will be assigned to g_2 according to case (c), since Child (Age 2 to 11) can only be assigned to g_2 and is located to the right of the group. Next, Adult will be assigned to the g_1 according to case (b). Note that Passengers will not be assigned to g_1 since g_1 has already been assigned a label.

Figure 2.9 gives the pseudo code of the label attachment algorithm, where AttachOne is shown at the bottom.

2.4 Empirical Evaluation

The proposed schema extraction solution has been evaluated with source query interfaces over varied domains. Our goal was to evaluate the performance of the *structure extraction* algorithm on capturing the grouping and ordering relationships of the attributes on the interfaces, and the effectiveness of the *label attachment* algorithm in finding the right labels for both the attributes and attribute groups. This section presents the experimental results in detail.

Data Set: All experiments were performed on the ICQ data set available from the UIUC Web integration repository [3]. The data set contains query interfaces to Deep Web sources in five domains: airfare, automobile, book, job, and real estate, with 20 query interfaces for each domain. Before the experiments, we

Domain	# Interfaces	# Attributes	Leaf Nodes			Internal Nodes			Depth		
			Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
Airfare	20	10.7	5	15	10.7	1	7	5.1	2	5	3.6
Auto	20	5.1	2	10	5.1	1	4	1.7	2	3	2.4
Book	20	5.4	2	10	5.4	1	2	1.3	2	3	2.3
Job	20	4.6	3	7	4.6	1	2	1.1	2	3	2.1
Real Estate	20	6.5	3	14	6.7	1	6	2.4	2	4	2.7

Table 2.2: Domains and characteristics of the data set

manually transformed the query interfaces in the data set into ordered-tree schemas, and used them as the gold standard to gauge the performance of the algorithms.

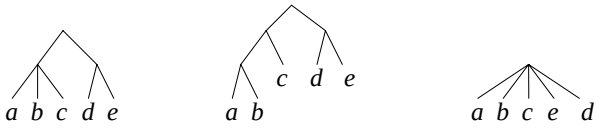
Table 2.2 shows the details of the data set. For each domain, columns 2–3 show the number of interfaces and the average number of attributes per interface in that domain. Columns 4–9 show the minimum, maximum, and average number of leaf nodes and internal nodes in the schema trees of the interfaces. And columns 10–12 show similar statistics on the depth of the schema trees.

2.4.1 Performance Metrics

For each interface, the schema tree produced by the structure extraction algorithm was compared with the schema tree in the gold standard with respect to their structures, i.e. the grouping and ordering of the attributes. A possible metric for comparing two trees is the *tree edit distance* [99], where the distance between two trees is taken to be the number of insertion, deletion, and relabeling operations necessary for transferring one tree into the other. But this metric does not sufficiently capture the *semantic* aspects of two trees, that is, the semantic closeness of two attributes in terms of their grouping relationships, and the relative semantics of two attributes in terms of their ordering relationships.

To address this challenge, we observe that the semantics of a schema tree can actually be encoded with the constraints which the schema enforces on its elements. In particular, we observe that the grouping and ordering relationships of attributes in the schema may be captured with *least-common-ancestor (LCA) constraints* and *precedence constraints*, to be formally defined below. Then, the *semantic* differences of two schema trees may be measured by the extent which the constraints from one schema tree are satisfied by the other schema tree.

Definition 6 (**LCA Constraint**) Consider a schema tree S and denote the lowest common ancestor of



(a) Schema S_1 (b) Schema S_2 (c) Schema S_3

Figure 2.10: Examples on constraints of schemas

two attributes (i.e., leaf elements) x and y in S as LCA(x, y). Consider three attributes x, y and z in S. We say that there exists a *LCA constraint* in form of $(x, y)z$ in S, if $\mathrm{LCA}(x, y) < \mathrm{LCA}(x, z)$ and $\mathrm{LCA}(x, y) < \mathrm{LCA}(y, z)$, where $n_1 < n_2$ denotes that element n_1 is a proper descendant of element n_2. □

Intuitively, the LCA constraint $(x, y)z$ indicates that two attributes x and y are semantically closer to each other than either to the attribute z. LCA constraints thus capture the semantic closeness of attributes expressed by the schema. It is interesting to note that given all the LCA constraints from an unordered schema tree S, S can be *fully* recovered in polynomial time [5].

Example 6 The LCA constraints in the schema S_1 shown in Figure 2.10.a are: $(a, b)d$, $(a, b)e$, $(a, c)d$, $(a, c)e$, $(b, c)d$, $(b, c)e$, $(d, e)a$, $(d, e)b$, and $(d, e)c$. □

Definition 7 (**Precedence Constraint**) Consider a schema S and a sequence of attributes, denoted as q_s, obtained from a *pre-order* traversal of S. We say that there exists a precedence constraint between two attributes x and y, denoted as $x \prec y$, in the schema S, if x appears *before* y in q_s. □

The precedence constraints thus capture the relative ordering of the attributes, both within the same group and across different groups.

Example 7 q_{S_1} is $<a, b, c, d, e>$. As such, some examples of the precedence constraints in S_1 are: $a \prec b$, $a \prec c$, $a \prec d$, and $d \prec e$. □

Based on these constraints, we evaluated the performance of schema extraction via *grouping metrics* and *ordering metrics* given as follows.

Grouping metrics: We measured the grouping performance of the structure extraction algorithm with three metrics: (LCA) precision, (LCA) recall, and (LCA) F1 [97]. Denote the schema tree for an interface

obtained by the structure extraction algorithm as S', and the schema tree given in the gold standard for the interface as S. The precision is then taken to be the percentage of the LCA constraints which are *correctly* identified by the algorithm (i.e., they are in both S' and S) over all the LCA constraints identified by the algorithm (i.e., they are in S'). And the recall is the percentage of the LCA constraints which are correctly identified over all the LCA constraints in S. F1 incorporates both precision and recall, computed as $2PR/(R+P)$.

Example 8 Suppose that S_1 in Figure 2.10.a is the schema tree given by the gold standard for an interface. Further suppose that S_2 in Figure 2.10.b is the schema tree given by the structure extraction algorithm for the same interface.

It can be verified that S_2 have all the nine LCA constraints in S_1 (see Example 6) plus an additional constraint $(a, b)c$. As such, the LCA precision of S_2 is 9/10 = .9, while the LCA recall of S_2 is 9/9 = 1. □

Ordering metrics: We measured the ordering performance of the structure extraction algorithm with two metrics: (precedence) precision and (precedence) recall. Since the number of precedence constraints in a schema with n attributes is always $n(n-1)/2$, precedence precision is always the same as precedence recall. In other words, precedence precision and precedence recall are both given by the ratio of the number of the precedence constraints correctly identified by the algorithm over $n(n-1)/2$.

Example 9 Suppose that S_1 in Figure 2.10.a is the schema tree given by the gold standard for an interface. Further suppose that S_3 in Figure 2.10.c is the schema tree given by the structure extraction algorithm for the same interface.

It can be verified that the only different precedence constraints in S_1 and S_3 are $d \prec e$ in S_1 vs. $e \prec d$ in S_3. As such, both the precedence precision and the precedence recall are 9/10 = .9 (note that since $n = 5$, $n(n-1)/2$ = 10). □

Performance Metrics for Label Attachment: We measured the performance of the label attachment algorithm on finding both attribute labels and group labels.

Attribute labeling metrics: The performance on finding attribute labels was measured with two metrics: (attribute labeling) precision and (attribute labeling) recall. The precision is the percentage of the correctly identified labels (i.e., labels attached to correct attributes) over all the labels identified by the algorithm.

Domains	Grouping		Ordering	Attribute Labels		Group Labels	
	Prec.	Rec.	Prec.(Rec.)	Prec.	Rec.	Prec.	Rec.
Airfare	93.3	95.1	99.5	95.0	93.7	97.7	92.3
Auto	92.0	92.9	99.7	99.2	97.6	100	89.5
Book	94.0	90.8	99.7	91.6	91.6	94.4	82.7
Job	82.4	95.2	97.7	95.1	94.9	100	89.7
Real Est.	92.1	95.6	96.0	99.1	97.3	100	90.5
Average	90.8	93.9	98.9	96.0	95.0	98.4	88.9

Table 2.3: The performance of the schema extractor

And the recall is the percentage of the correctly identified labels over all the attribute labels given in the gold standard.

Group labeling metrics: Similarly, the performance on finding group labels was measured with two metrics: (group labeling) precision and (group labeling) recall. But since the groups identified by the structure extraction might not be always correct, a more accurate way of evaluating the group labeling is to base it on the attributes. In particular, for each attribute on the interface, we associate with the attribute the labels of all groups which contain the attribute.

Example 10 The group labels associated with the attribute f_3 in Figure 2.1.c are Departure Date and When Do You Want to Go?. Intuitively, the group labels associated with an attribute, together with the label of the attribute, denote to the users what the attribute means. □

Based on the above discussions, the group labeling precision is taken to be the percentage of the group labels correctly associated with the attributes by the algorithm over all the group labels associated with the attributes by the algorithm (note that typically a group label may be associated with more than one attributes). And the recall is taken to be the percentage of the group labels correctly associated with the attributes by the algorithm over all the group labels associated with the attributes in the gold standard.

2.4.2 The Performance of the Structure Extraction Algorithm

Columns 2–4 of Table 2.3 show the performance of the structure extraction algorithm.

Grouping: Columns 2–3 show the performance of the structure extraction algorithm on discovering the grouping relationships of the attributes over the five domains. We observe that the precisions range from 82.4% in the job domain to 94.0% in the book domain, with an average of 92.1%. Note that the job domain is the only domain whose precision is lower than 90%. Detailed analysis indicated that some interfaces in

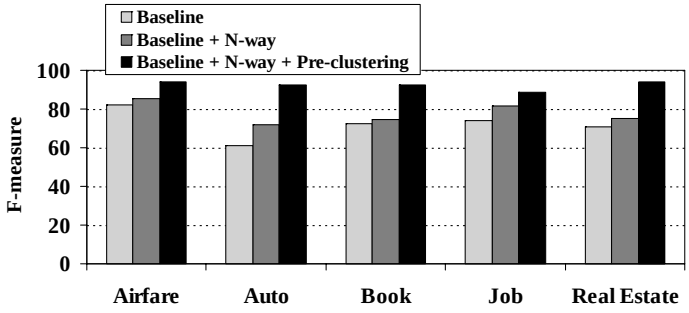


Figure 2.11: Effects of the n-way clustering and pre-clustering

this domain use shaded area to indicate attribute groups. And since some of the attributes in these groups are actually farther away from each other than from the attributes not from the same group, as a result, several attribute groups found by the algorithm were incorrect. A possible remedy is to introduce a new grouping pattern to recognize the attribute groups which are delimited with shaded areas.

We further observe that the recalls range from 90.8% in the book domain to 95.6% in the real estate domain, with an average of 93.9% over the five domains. These indicate that the algorithm is highly effective in identifying the grouping relationships of the attributes.

We also examined the effects of the n-way clustering and pre-clustering on the grouping performance. Figure 2.11 shows the results. For each domain, the three bars (from left to right) represent the performance produced respectively by the algorithms without the n-way clustering and pre-clustering, with only the n-way clustering, and with both the n-way clustering and pre-clustering being incorporated. All the performances are measured by F1. Note that the last bars correspond to the figures shown in Table 2.3.

It can be observed that with the n-way clustering, the performance improved consistently over the domains, with the largest increase (10.4 percentage points) in the auto domain. Furthermore, the pre-clustering significantly improved the performance in all five domains, ranging from 6.7 percentage points in the job domain to as high as 20.8 percentage points in the auto domain. These indicate the effectiveness of both the n-way clustering and pre-clustering.

Ordering: Column 4 shows the performance of the structure extraction algorithm in identifying the ordering of attributes over the five domains. It can be observed that the accuracy ranges from 96% in the real estate domain to as high as 99.7% in both the auto and book domains. This indicates that the algorithm is highly effective in determining the ordering of the attributes.

2.4.3 The Performance of the Label Attachment Algorithm

The last four columns of Table 2.3 show the performance of the label attachment algorithm.

Attribute labeling: Columns 5–6 show the performance on attribute labeling. We observe that the precisions range from 91.6% in the book domain to as high as 99.2% in the auto domain, with an average of 96% over the five domains, and that the recalls range from 91.6% to 97.6%, with an average of 95%. These indicate that the label attachment algorithm is highly accurate in determining the labels of the attributes.

Group labeling: The last two columns show the performance on group labeling. It can be observed that high precisions are achieved over the five domains, with 94.4% in the book domain, 97.7% in the airfare domain, and perfect precisions in the other three domains.

It can be further observed that the recalls range from 82.7% in the book domain to 92.3% in the airfare domain. We examined the book domain which had the relatively low recall. Detailed results indicated that there are several interfaces where some of the groups identified by the structure extraction algorithms are only partial, that is, they do not contain all the attributes in the group. As a result, the label attachment assigns the group label to the partial group, resulting in the low recall. This indicates that the label attachment algorithm could be very sensitive to the performance of the structure extraction algorithm, which is not surprising.

Overall, the average precision 98.4% and the average recall 88.9% were achieved on the five domains. These indicate that the label attachment is very effective in identifying group labels.

2.4.4 Discussion

We have shown that the proposed schema extraction algorithm can effectively infer the structure of query interface from the spatial placement of attributes on the interface. But it is possible that some query interfaces may simply list its attributes one after another, with no explicit group delimiters such as white space and group label. We say that the structure of these query interfaces is *implicit*. Currently, the schema extractor does not handle query interfaces with implicit structure.

For example, Figure 2.12 shows a query interface to a car sale source. The interface contains six attributes which fall into three categories: the first two attributes (i.e., Zip Code and Distance) concern the location of car dealership, the third attribute is on the price, and the last three attributes indicate the specification of the car. But note that the interface does not explicitly separate the attributes in different

Figure 2.12: A query interface with implicit structure

groups. So it is very difficult for the schema extractor to infer the relationships among the attributes without understanding the semantics of each attribute.

2.5 Summary

In this chapter, we proposed hierarchical modeling of query interfaces and presented an effective approach to extracting the hierarchical schema of query interface from its representation in HTML form. The key novelties of our solution are: (1) capturing both the grouping and ordering relationships of attributes; (2) employing a novel spatial clustering-based algorithm to exploit the placement of attributes on the interface for structure extraction; and (3) developing new constraint-based metrics for evaluating the performance of schema extraction.

Since schema extraction is the first step in interface integration, its performance is of crucial importance to the remaining steps in the integration. Indeed, as we will see in Chapter 3, schema matcher extensively exploits the grouping and ordering relationships of attributes to accurately identify the mappings among the attributes. And as Chapter 5 will show, the quality of global interface constructed by the schema merger largely depends on whether the schemas obtained by the schema extractor have accurately captured the structure of source interfaces.

Chapter 3

Schema Matching

3.1 Introduction

Given a set of interface schemas, the *schema matcher* must accurately identify the semantic correspondences (i.e., mappings) among the attributes from different interfaces. We consider both simple and complex mappings of attributes. In this section, we examine each type of mappings, discuss the challenges in identifying these mappings, and give an overview of our solution.

3.1.1 Simple Mappings

A simple mapping is a *1:1* semantic correspondence between two attributes on different interfaces. For example, two sources in the automobile domain may both have an attribute for the make of vehicles on their query interfaces.

The major challenge to the identification of 1:1 mappings of attributes is the *label mismatch* problem. Label mismatches occur when similar attributes from different interfaces are annotated with very different labels. For example, consider query interfaces in the airfare domain. Attributes for the service class may be annotated with varied labels such as class of service, class of ticket, class, cabin, preferred cabin, and flight class.

Typically, the label of an attribute may be either a single word (e.g., class), a phrase (e.g., class of service), or even a sentence. Labels of similar attributes may often have common words (e.g., class in flight class and class of ticket) or they may be synonyms (e.g., cabin and class). But note that a general-purpose semantic lexicon such as WordNet [32] does not help much in the identification of domain-specific synonyms. For example, it is difficult to infer from WordNet that cabin is synonymous to class in the airfare domain. Furthermore, domain-specific lexicons are not generally available and they can be expensive to build.

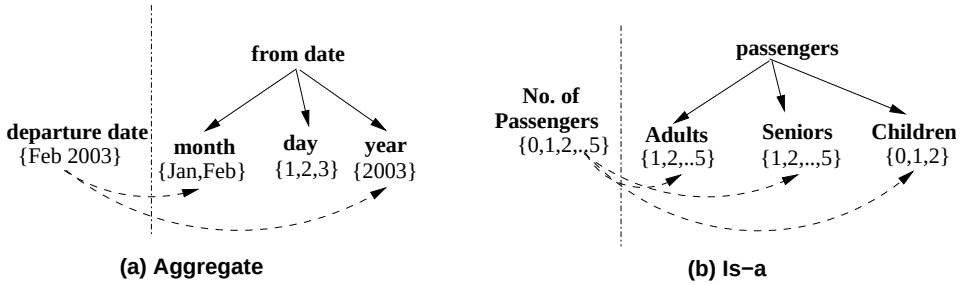


Figure 3.1: Examples of 1:m mappings

3.1.2 Complex Mappings

While 1:1 mappings account for the majority of attribute mappings, we found that 1:m mappings occur in every domain we studied and very frequently in some domains (see also Section 3.4). A *1:m* mapping is a mapping between an attribute on one interface and *multiple* attributes on another interface. Most of the current solutions [41, 43] consider only the 1:1 mappings, thus largely simplifying the problem.

We observe that there may be two types of 1:m mappings among the attributes: *aggregate* and *is-a*. For both types, attributes on the many side of the mapping refine the attribute on the one side. But there is a major difference. In the aggregate type, the value of an attribute on the many side is typically only a *part of* the value of the attribute on the one side; while in the is-a type, the value of the attribute on the one side is typically the *union* (or sum) of the values of the attributes on the many side. To illustrate, Figure 3.1 shows examples on both types of 1:m mappings among the attributes in the airfare domain. Note that for the attributes on the many side, we also show their parent node. The reason for this will become clear below.

Finding 1:m mappings is even more challenging than finding 1:1 mappings. Note that domain-specific concept hierarchies might help but again they are rarely available and also expensive to construct manually.

3.1.3 Toward a Highly Accurate Attribute Matching

Handling 1:1 mappings: To cope with the label mismatch problem, we exploit the "bridging" effect in the process of matching attributes from a large number of query interfaces at once. Note that the bridging effect is in spirit similar to the idea of reusing existing mappings [83]. In Section 3.2, we propose a clustering-based matching algorithm which exploits exactly this observation to effectively identify 1:1 mappings of the attributes. The following example further illustrates the bridging effect.

Example 11 Consider three attributes, e, f and g, from three different interfaces in the computer hardware

domain. Suppose that label(e) = cpu, dom(e) = {celeron, pentium, duron}, label(f) = processor, dom(f) may contain any strings, label(g) = processors, and dom(g) = {athlon, celeron, pentium, xeon}. Further, suppose that the three attributes have totally different names. We note that neither the labels nor the domains of e and f are similar. Nevertheless, the domain of e is similar to the domain of g while the label of f is similar to the label of g. As a result, attribute g may serve as a potential "bridge" to help match attributes e and f. □

Handling 1:m mappings: To cope with the lack of domain-specific concept hierarchies, we exploit the following observations to help identify 1:m mappings of attributes. (1) *Value correspondence:* As described above, the values of the attributes involved in a 1:m mapping are often suggestive of the mapping. (2) *Placement proximity:* Attributes on the many side of a 1:m mapping are typically very close to each other on the interface. (3) *Label similarity:* The label of the attribute on the one side of a 1:m mapping often bears similarity with the *parent* label of the attributes on the many side of the mapping.

Based on these observations, Section 3.2.3 describes a three-staged approach which extends the clustering algorithm to effectively discover 1:m mappings of attributes.

User interactions: Like other schema matching approaches [43, 64], the matching solution proposed above has several parameters which require manual tunings. To address this challenge, Section 3.3 proposes several approaches to interactively learning the parameters and resolving uncertain mappings. Our experiments (Section 3.4) indicate that the proposed approaches significantly improve the matching accuracy with only a small amount of user interaction.

3.2 Attribute Matching via Clustering

3.2.1 Attribute Similarity Function

As proposed in Section 2.1, an attribute may be characterized by its three properties: name, label, and domain. Intuitively, if two attributes are semantically similar, they should have similar properties. As such, the (aggregate) similarity between two attributes e and f, denoted as $\mathcal{AS}(e, f)$, is given as follows:

$$\mathcal{AS}(e, f) = \lambda_{ls} * \text{lingSim}(e, f) + \lambda_{ds} * \text{domSim}(e, f), \tag{3.1}$$

where lingSim is the linguistic similarity based on their names and labels, domSim is the domain similarity based on their domains, and λ_{ls} and λ_{ds} are weight coefficients, indicating the relative importance of the two component similarities.

The Linguistic Similarity

The name and label of an attribute can be regarded as two description-level properties of the attribute. Two attributes are linguistically similar if they have similar names and labels.

Fundamental to the computation of linguistic similarities is the measure on the similarity between two strings (of words). For this, we employ the Cosine function commonly used in Information Retrieval [19, 89]. Specifically, the Cosine similarity of two strings can be obtained as follows. Consider two strings s and p. First, stop words (i.e., non-content words) such as "the", "a", and "an", are removed from each string. Suppose that the resulted strings contain m distinct content words (i.e., terms). We may then represent each string as an m-dimensional vector of terms with weights, e.g., $\vec{s} = (w_1, w_2, \ldots, w_m)$, where w_i is the number of occurrences of the i-th term t_i in the string s. Finally, the similarity of the two strings, s and p, is computed as: $\mathrm{Cosine}(\vec{s}, \vec{p}) = \vec{s} \cdot \vec{p}/(\|\vec{s}\| * \|\vec{p}\|)$. In the following, to simplify notations, we will denote $\mathrm{Cosine}(\vec{s}, \vec{p})$ as $\mathrm{Cos}(s, p)$. In the experiments, words in the strings are also stemmed [79].

The linguistic similarity of two attributes e and f, denoted as lingSim(e, f), is then a weighted average similarity of their names, labels, and name vs. label:

$$\mathrm{lingSim}(e, f) = \lambda_n * \mathrm{nSim}(e, f) + \lambda_l * \mathrm{lSim}(e, f) + \lambda_{nl} * \mathrm{nlSim}(e, f), \tag{3.2}$$

where nSim(e, f) = Cos(name(e), name(f)), lSim(e, f) = Cos(label(e), label(f)), and nlSim(e, f) = max{Cos(name(e), label(f)), Cos(name(f), label(e))}.

Normalization: As noted in Section 2.1, attribute names may often contain concatenated words and abbreviations. As such, they need to be normalized before they are used to compute the linguistic similarities. We apply the following normalizations [64, 93].

(a) *Tokenizations* are used to cope with concatenated words. First, delimiters and case changes are used to suggest breakdowns of the concatenated words. For example, "departCity" into "depart City" and "first_name" into "first name". Next, words appearing in the labels are utilized to suggest further break-

downs. Specifically, a domain dictionary is constructed automatically for this purpose. The dictionary contains all the words appearing in the labels over the interfaces. For example, "deptcity" will be split into "dept" and "city" if the word "city" appears in the dictionary.

(b) *Transformations* are used to expand the abbreviations. For example, "dept" into "departure". We again utilize the domain dictionary constructed above to suggest the expansions. To avoid false expansions, we require that the word w to be expanded meets the following conditions: (1) w is not in the dictionary; (2) w contains at least three letters; and (3) the first letter of w is the same as that of the expanding word.

The Domain Similarity

The domain similarity of two attributes e and f, denoted as domSim(e, f), is the similarity of their domains: dom(e) and dom(f). We will also use domSim to denote the similarity of two domains.

Simple Domains: We start by considering simple domains. A simple domain is a domain whose values contain only one component. Simple domains can be of varied types. We consider the following types of simple domains: *time*, *money*, *area*, *calendar month*, *int*, *real*, and *string*. Since the domain type of an attribute is not specified on the interface, we infer it from the values in its domain. The inference is done by matching the values against a set of regular-expression patterns, one for each domain type, specifying the format of the values in the domain. For example, the pattern for the *time* domains may be "[0-9]{2}:[0-9]{2}", which recognizes values such as "03:15".

In addition to the values in the domain, the label of an attribute might also contain information on the domain type of the attribute. For example, an attribute whose label contains '$' sign or keyword "USD" is likely to be the *money* type. For the attributes which we are not able to infer their domain types, we assume their domains are the *string* type with an infinite cardinality.

Similarity of Two Simple Domains: The similarity of two simple domains d and d', denoted as domSim(d, d'), is evaluated based on both their types and values as follows:

$$\text{domSim}(d, d') = \lambda_t * \text{typeSim}(d, d') + \lambda_v * \text{valueSim}(d, d'). \tag{3.3}$$

If two domains have different types, their similarity is set to zero. Otherwise, their typeSim is 1 and we further evaluate their value similarity. The value similarity is evaluated differently for character domains

(whose values are strings) and numeric domains (whose values are numbers).

Consider two character domains d and d'. Suppose that d = $\{u_1, u_2, ..., u_m\}$ and d' = $\{v_1, v_2, ..., v_n\}$, where u_i's and v_j's are strings. With a desired threshold τ, we first obtain pairs of similar values, one from each domain, via a BEST-MATCH procedure: (1) Evaluate the Cosine similarity for all pairs of values. (2) Select the pair with the maximum similarity among all pairs. Denote the pair as (x, y), where $x \in d$ and $y \in d'$. (3) Remove all pairs which contain either x or y from further considerations. (3) Repeat steps 2–3 on remaining pairs until none of the pairs has a similarity $> \tau$. Let the set of chosen pairs be C. The valueSim(d, d') is then given by the Dice's function [23, 24] as $\frac{2*|C|}{|d|+|d'|}$.

The value similarity of two numeric domains h and h' is evaluated based on the extent by which the value ranges of their domains overlap. Specifically, valueSim(h, h') is given by:

$$\frac{\min\{\max(h), \max(h')\} - \max\{\min(h), \min(h')\}}{\max\{\max(h), \max(h')\} - \min\{\min(h), \min(h')\}},$$

where min(x) and max(x) respectively give the minimum and maximum of the values in the domain x. Note that the numerator is the range where the two domains overlap and the denominator is the outer span of the two domains. It is easy to see that for two identical domains, their value similarity is 1. If two domains do not overlap, their value similarity is set to zero. For discrete numeric domains such as *int*, the above formula might underestimate their value similarity. For these domains, we adjust the formula by adding constant 1 to both the numerator and denominator but only when the numerator is greater than zero.

For the attributes whose domains are the *string* type with an infinite cardinality, we assume that their domains are dissimilar to the domains of any other attributes.

3.2.2 Finding 1:1 Mappings

We employ a hierarchical agglomerative clustering algorithm [49] to identify 1:1 mappings of attributes. The algorithm is shown in Figure 3.2. It expects three inputs: a set of interfaces $\mathcal{S}$, the similarity matrix M for the attributes in $\mathcal{S}$, and a stopping threshold $\tau_c \geq 0$. The algorithm outputs a partition over the attributes in $\mathcal{S}$ such that similar attributes are in the same partition while attributes in different partitions are dissimilar.

The similarity matrix M is obtained as follows. Suppose that the total number of attributes over the given interfaces is m. For every two attributes from different interfaces, their aggregate similarity is evaluated using Formula 3.1. The results are stored in an $m \times m$ symmetric matrix M, where the entry $M[i, j]$ gives

CLUSTER($\mathcal{S}$, M, τ_c) $\rightarrow P$:
Input: $\mathcal{S}$, a set of interfaces; M, the similarity matrix for the attributes in $\mathcal{S}$;
τ_c, a stopping threshold.
Output: P, a partition over the attributes in $\mathcal{S}$

1. Place each attribute in $\mathcal{S}$ in a cluster by itself
2. While there are two clusters with similarity $> \tau_c$
   a. choose two clusters c_i and c_j with the maximum similarity
   b. resolve ties if necessary
   c. merge c_i and c_j into a new cluster c_k, remove clusters c_i and c_j
   d. remove all rows and columns associated with c_i and c_j in M,
      add a new row and column for c_k
   e. compute the similarities of c_k with other clusters using Formula 3.4
3. Return final clusters of attributes

Figure 3.2: The clustering algorithm for finding 1:1 mappings

the aggregate similarity between the i-th and j-th attributes. Note that the entries for two attributes from the same interface are set to zero.

We now describe the algorithm in detail. In a nutshell, the algorithm starts by placing each attribute in a cluster by itself, then repeatedly merges two clusters with the maximum similarity until no two clusters have a similarity greater than τ_c.

Greedy Matching: Typically, each attribute may have a large number of candidate mappings and we have to decide which ones are the "best". The matching problem well-studied in graph theory [62] provides a rich set of possible criteria such as: maximum cardinality, maximum total weight, and stable marriage. A matching has a maximum cardinality if it has the largest number of mappings; a matching has a maximum total weight if the sum of the weights of its mappings is the largest; and a matching is stable if it does not have two mappings (x, y) and (x', y') such that x prefers y' to y and y' prefers x to x'.

Empirical studies in [70] show that the *perfectionist egalitarian polygamy* selection metric (i.e., no male or female is willing to accept any partner(s) but the best) produces the best results in a variety of schema matching tasks. The *greedy choice* step of the clustering algorithm can be regarded as a monogamy version of this metric.

Specifically, each merging iteration of the clustering process involves the following two stages. (1) *Greedy choice:* Two clusters c_i and c_j with the *maximum* similarity are chosen to merge into a new cluster c_k. All attributes in c_k are considered to be 1:1 mapped to each other. This stage corresponds to steps 2(a–c). (2) *Constraint enforcement:* Consider two attributes $f_i, f_j \in c_k$ and suppose that f_i is from interface

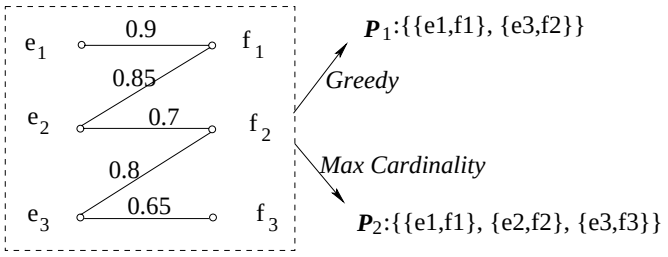


Figure 3.3: An example on finding attribute mappings

S_u while f_j from interface S_v. Since we only consider 1:1 mappings, f_i can not be mapped to any other attributes in S_v while f_j can not be mapped to any other attributes in S_u. This stage corresponds to steps 2(d–e).

It can be shown that the matching obtained by the above greedy strategy is stable. The following example further illustrates the greedy strategy by contrasting it with the maximum cardinality criterion.

Example 12 Consider interface E with attributes e_1, e_2, and e_3, and interface F with attributes f_1, f_2, and f_3. Suppose that their aggregate similarities are as shown in the similarity graph in Figure 3.3. Further suppose that the clustering threshold $\tau_c = .6$.

When the clustering starts, there are six clusters, each for an attribute on the interfaces. For example, f_2 is in cluster $\{f_2\}$. The greedy strategy will first choose to merge $\{e_1\}$ and $\{f_1\}$ since they now have the maximum similarity. Note that the new cluster $\{e_1, f_1\}$ can not be merged with cluster $\{e_2\}$, otherwise 1:1 mapping constraint will be violated. As such, $\{e_3\}$ and $\{f_2\}$ will be merged next since they have the maximum similarity (.8).

It can be verified that the final partition is P_1. In contrast, the maximum cardinality criterion yields a different partition P_2. Note that the number of mappings in P_2 is 3, the largest possible for this example, while the number of mappings in P_1 is 2. On the other hand, P_1 is stable, while P_2 is not. □

The Cluster Similarity: Consider cluster c_i = $\{c_{i_1}, c_{i_2}, ..., c_{i_m}\}$ and cluster c_j = $\{c_{j_1}, c_{j_2}, ..., c_{j_n}\}$. The similarity between c_i and c_j, denoted as $\mathrm{Sim}(c_i, c_j)$, is then given by:

$$\max_{1 \le u \le m, 1 \le v \le n} \{\mathcal{AS}(c_{i_u}, c_{j_v})\}. \tag{3.4}$$

Note that if c_i and c_j contain attributes from the same interface, then $\mathrm{Sim}(c_i, c_j) = 0$. Formula 3.4 yields a single-link algorithm.

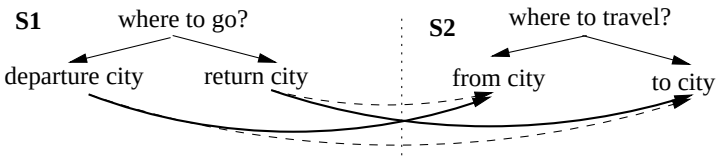


Figure 3.4: An example on tie resolution

Ordering-based Tie Resolution: Ties occur when a cluster has the maximum similarity with multiple clusters. The main reason for the ties is that aggregate similarities alone may not be sufficient for identifying the correct mappings. Ties frequently occur in some domains (see our experiments for more details), thus it is important to properly resolve them.

We resolve ties by exploiting the ordering of the attributes in the clusters involved in the ties. Specifically, consider a cluster c which has the maximum similarity with a set s of other clusters. First, we select two *reference* interfaces S_1 and S_2. S_1 can be any interface whose attributes appear in some cluster in s other than c, and S_2 can be any interface whose attributes are in the cluster c.

Next, we find two reference attributes e and f, where e is from S_1 and f is from S_2, such that e and f have the maximum similarity, and no other attributes appearing before e on S_1 have the maximum similarity with f and vice versa (i.e., e and f are each other's first best choice). If found, we merge the cluster which contains e with the cluster which contains f. Otherwise, we randomly merge two clusters with the maximum similarity.

Example 13 Consider two reference interfaces in the airfare domain as shown in Figure 3.4, where four pairs of attributes have the maximum similarity. Attributes departure city and from city will be selected as the reference attributes. Note that this corresponds to the intuition that normally departure information appears before returning information on an airfare interface. □

3.2.3 Finding Complex Mappings

The clustering-based matching algorithm proposed in Section 3.2.2 considers only the 1:1 mappings. In this section, we extend the algorithm so that it also handles 1:m mappings. The complete attribute matching algorithm is shown in Figure 3.5. It consists of three phases: (1) *preliminary-1-m-matching phase*, which exploits the properties of attributes and the structure of interfaces to identify an initial set of 1:m mappings; (2) *1-1-matching phase*, which employs the clustering algorithm (Figure 3.2) to discover 1:1 mappings

among the attributes which are not on the one side of identified 1:m mappings; and (3) *final-1-m-matching phase*, which infers additional 1:m mappings from the mappings found at the first two phases.

We now describe the algorithm in detail, starting with a necessary definition.

Definition 8 (Composite Domain & Attribute) A domain d is composite if each value in d is a k-tuple: $<t_1, t_2, ..., t_k>$, where t_i is a value from the i-th sub-domain of d, denoted as d_i. All d_i's are simple domains. k is the arity of d, denoted as $\phi(d)$. an attribute is composite if its domain is composite. □

The domain type of a composite domain is also composite, consisting of an ordered list of simple domain types, one for each of its sub-domains. A special composite domain type *date* is pre-defined for calendar dates.

To determine if an attribute is composite, we employ a *structure extraction* process similar to that in data cleaning [85]. In particular, delimiters in the values of the domain are used to suggest the structure of the domain. The delimiters may include punctuation characters, white spaces, and special words such as "to". For an attribute to be composite, we require that the overwhelming majority of the values in its domain can be consistently decomposed into the same number of components.

For the attributes which do not have any instances, we also look in their labels for possible information on their domains. For example, "mm/dd/yyyy" and "mm/dd/yy" are commonly used to annotate attributes of the *date* type.

Similarity of Composite vs. Simple/Composite Domains: Consider two domains d and d', at least one of which is a composite domain. In other words, either $\phi(d) > 1$, or $\phi(d') > 1$, or both. The similarity of d and d' is then evaluated by the extent their sub-domains are similar. Specifically, since sub-domains are all simple domains, their similarity is evaluated using Formula 3.3, and two sub-domains are determined to be similar if their similarity exceeds a threshold τ'. We then employ the BEST-MATCH procedure to determine a set of pairs of similar sub-domains: $\{(d_i, d'_j)|\mathrm{domSim}(d_i, d'_j) > \tau'\}$, denoted as C'. Finally, the similarity of d and d' is evaluated via the Dice's function as: $\frac{2*|C'|}{\phi(d)+\phi(d')}$.

Identify a Preliminary Set of 1:m Mappings

The *preliminary-1-m-matching* phase discovers an initial set of 1:1 mappings of both aggregate and is-a types.

ATTRIBUTEMATCH(S) $\rightarrow P$ and Q'
Input: S, a set of query interfaces
Output: P and Q', respectively 1:1 and 1:m mappings of attributes in S

1. Discover initial 1:m mappings:
   $Q \leftarrow$ IDENTIFYINITIALONETOMANYMAPPINGS(S)
2. Find 1:1 mappings:
   a. Obtain similarity matrix:
      $M \leftarrow$ COMPUTEPAIRWISESIMILARITIES(S)
   b. Identify 1:1 mappings via clustering:
      $P \leftarrow$ CLUSTER(S, M, τ_c)
3. Infer additional 1:m mappings:
   $Q' \leftarrow$ OBTAINFINALONETOMANYMAPPING(P, Q)

Figure 3.5: The attribute matching algorithm

Aggregate Type: To identify an initial set of aggregate 1:m mappings, we proceed as follows. Consider all attributes on the interfaces. For each attribute e on interface S, we first check if it is a composite attribute as described above. If e is composite, then on every interface other than S, denoted as X, we look for a set of attributes $\mathbf{f} = \{f_1, f_2, \ldots, f_n\}$ where $n > 1$, which meets the following three conditions. (a) f_i's are siblings, that is, they share the same parent p but $\mathbf{f}$ might be a proper subset of the set of all children of p. For example, day is not involved in the 1:m mapping shown in Figure 3.1.a. Note that this condition essentially exploits the *placement proximity* observation in Section 3.1.3. (b) The label of p is highly similar to the label of e. (c) There is a subset s of sub-domains of the domain of e, such that there is a 1:1 correspondence between each sub-domain in s and the domain of some attribute f_j (or sub-domain if f_j is composite) in $\mathbf{f}$, in the sense that they have high similarity (according to Formula 3.3). Note that this condition essentially reflects the *value correspondence* observation in Section 3.1.3.

If there exists such an $\mathbf{f}$ on the interface X, a 1:m mapping of aggregate type is then identified between e and attributes in $\mathbf{f}$, denoted as $e \leftrightarrow \{f_1, f_2, \ldots, f_n\}$.

Is-a Type: The identification of aggregate 1:m mappings requires detecting composite attributes and discovering corresponding sub-attributes. Typically, the domains of the sub-attributes may be different. In contrast, the identification of is-a 1:m mappings requires that the attribute on the one side is a non-composite attribute and that the domain of each sub-attribute is similar to that of the attribute on the one side.

Specifically, for each non-composite attribute e', we check if there exists a set of attributes $\mathbf{f} = \{f_1, f_2, \ldots, f_n\}$ where $n > 1$, on another interface X, which meets the following conditions. (a) All f_i's are siblings and their parent p does *not* have any children other than f_i's. (b) The label of p is highly similar to the label of

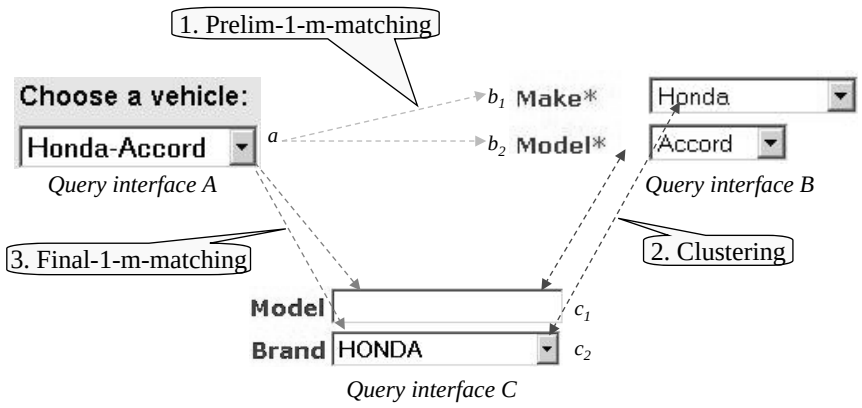


Figure 3.6: An inference example

e. (c) The domain of *each* f_i is highly similar to that of e.

If there exists such an $\mathbf{f}$ on the interface X, a 1:m mapping of is-a type is then identified between e' and attributes in $\mathbf{f}$, denoted as $e' \leftrightarrow \{f_1, f_2, \ldots, f_n\}$.

Dealing with Infinite Domains: As discussed earlier, there are some attributes which we are not able to infer their domain types and assume that their domains are the string type with an infinite cardinality. Since the similarity between such domain and any domain is zero, the above approaches can not be applied.

To cope with this, we introduce an additional approach which utilizes the labels to identify possible 1:m mappings. Specifically, consider all attributes which are not involved in any 1:m mappings identified above. For each such attribute g, we seek a set of *sibling* attributes $\mathbf{f} = \{f_1, f_2, \ldots, f_n\}$ where $n > 1$, which meets at least *one* of the following conditions. (1) f_i's are the only children of their parent p, and the label of g is *identical* to the label of p. (2) The label of g can be decomposed into several components with ',', '/', 'or' as delimiters, and the label of each f_i is one of these components.

Obtain the Final 1:m Mappings of Attributes

Our experiments show that the mappings identified in the preliminary-1-m-matching phase are quite accurate (i.e., with high precision). But there are cases where direct evidences between the attributes involved in a 1:m mapping might not be sufficient to meet the required conditions as described above. As a result, these mappings fail to be identified, reducing the recall. To cope with this, in the *final-1-m-matching* phase, an inference process is carried out where the 1:m mappings identified in the premilinary-1-m-matching phase are combined with the 1:1 mappings identified in the clustering process to infer additional 1:m mappings.

We also require that the attributes on the many side of 1:m mappings are siblings. We use the following example to illustrate this inference process.

Example 14 Consider three interfaces shown in Figure 3.6. Suppose the preliminary-1-m-matching phase is able to identify a 1:m mapping $a \leftrightarrow \{b_1, b_2\}$, where attribute a (Choose a vehicle) comes from interface A, and both attributes b_1 (Make) and b_2 (Model) from interface B. The identification exploits the value correspondence among the attributes.

Suppose further that the clustering process discovers two 1:1 mappings: $b_1 \leftrightarrow c_1$ and $b_2 \leftrightarrow c_2$, where both c_1 (Model) and c_2 (Brand) are from interface C. Then, a new 1:m mapping $a \leftrightarrow \{c_1, c_2\}$ will be inferred by the final-1-m-matching phase. Note that it is difficult to directly identify this mapping since attribute c_1 does not have any instances. □

3.3 User Interaction

As typical of schema matching algorithms, the proposed attribute matching algorithm requires a set of parameters to be manually set. Since these parameters may be domain-specific, when the system is applied to a different domain, a different set of parameters may need to be used. Typically, these parameters are tuned in a trial-and-error fashion with no principled guidance. Furthermore, since there might not exist a set of *best* parameters, errors may still occur: (1) some mappings might fail to be identified (i.e., *false negatives*); and (2) some identified mappings are not correct (i.e., *false positives*).

In this section, we describe several approaches to address the above challenges. We first propose an approach to learning the parameters by asking user selective mapping questions. We further present several approaches to interactively resolving uncertain mappings by posing mapping questions to the user. For each uncertain mapping, we provide the user with the attributes involved in the mapping, showing both the labels and instances of the attributes, and the user only needs to give "yes"/"no" responses.

3.3.1 Parameter Learning

We observe that the attribute similarity, denoted as fs, is actually a linear combination of several component similarities, that is, $\mathrm{fs} = a_1 * \mathrm{cs}_1 + a_2 * \mathrm{cs}_2 + \cdots + a_n * \mathrm{cs}_n$, where a_i is the weight coefficient indicating the relative importance of the i-th component similarity cs_i. As such, the attribute matching algorithm may

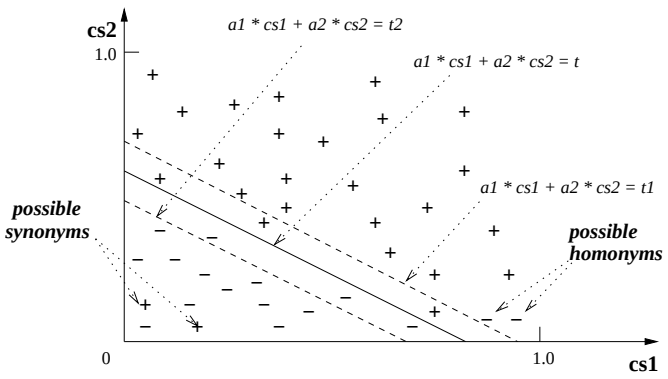


Figure 3.7: Thresholding function

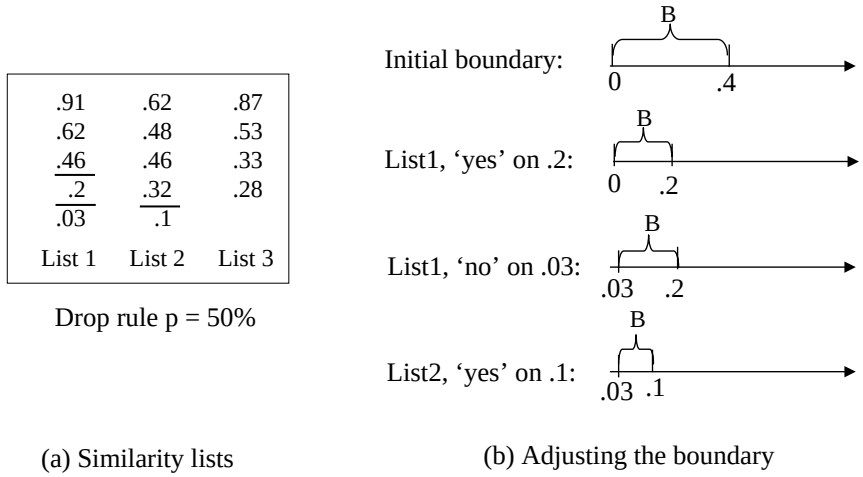
be regarded as a thresholding function $\mathrm{fs} > \tau$, or equivalently, $a_1 * \mathrm{cs}_1 + a_2 * \mathrm{cs}_2 + \cdots + a_n * \mathrm{cs}_n > \tau$. In other words, two attributes will be judged to be similar only if their similarity $\mathrm{fs} > \tau$.

Consider a simple case where the attribute similarity fs has only two component similarities, cs_1 and cs_2. Figure 3.7 plots the distribution of attribute similarities in two dimensions, one for each component similarity. A point is shown in '+' sign if two attributes with the corresponding component similarity indeed match; and in '-' sign otherwise. If the component similarity functions are reasonably accurate in capturing the similarities of attributes, we expect a typical distribution as shown in the figure. That is, matching attributes would normally have at least one large component similarity while non-matching attributes would have low values for both of their component similarities. Clearly, a good thresholding function should be such a dividing line that the majority of positive points are located *above* and the majority of negative points located *below*.

There are many different ways of learning the thresholding function. Here, we propose an approach for learning the threshold τ, while a_i's are set to some *domain-independent* constants.

Learning the Threshold: We observe that there can only be two possibilities for each attribute on an interface: either it matches with *some* attribute on some other interface, or it does not match with *any* attribute on any other interface. Assuming that the similarity function is reasonably accurate, we will have relatively larger similarity values for the attributes in the former case, and relatively smaller similarity values for the attributes in the latter case. In other words, there will be a gap between these two types of similarity values and a good threshold can be set to any value within this gap.

Motivated by this observation, we propose an approach to determining a boundary for the good thresholds. Specifically, we proceed as follows. We first set the boundary to some reasonable range $[a, b]$. We then



(a) Similarity lists (b) Adjusting the boundary

Figure 3.8: An example on threshold learning

consider all the interfaces in turn. For each interface, we apply the following steps to update the boundary: (1) For each attribute on the interface, obtain the *maximum* similarity of the attribute with other attributes on all other interfaces. (2) Arrange these similarities into a list by the descending order of their values. (3) Start from the first value in the list which is within the current boundary, and for each such value v, examine if v is significantly lower, say by the percent p, than the previous value in the list. (4) If yes, ask the user to determine if the pair of attributes corresponding to v is matching. (5) If the answer is yes, lower the upper bound to v and continue on the list; if the answer is no, increase the lower bound to v and stop further processing down the list.

Example 15 Consider a set of three query interfaces. Suppose the obtained similarity lists are as shown in Figure 3.8.a, one list for each interface. Further, suppose that the drop rule p = 50% and initially, the boundary B is [0, .4].

Figure 3.8.b illustrates the process of interactively adjusting the boundary based on these lists. Start from the first list. Note that the first drop of more than 50% occurs at .2 (from .46). So the user is asked to confirm the mapping between the attributes whose similarity is .2. For this, users are presented with all the information on the involved attributes and their schemas. Suppose that the user confirms this mapping. In other words, two attributes would still be considered to be matching if their similarity value is .2. So the upper bound of the boundary is adjusted from .4 to .2. The next question is on .03. Suppose that the user indicates that the corresponding mapping is incorrect, the lower bound of the boundary is then adjusted from 0 to .03.

It can be verified that after repeating similar process on the remaining two lists, the final boundary is [.03, .1]. Therefore, the matching threshold may be set to a value between this boundary. □

Denote the updated boundary resulted from the above process as [a', b']. If this boundary is still too rough, a refinement process is carried out. For this process, we reconsider all the lists of maximum similarities obtained above, one from each interface, and select the list with the largest number of values within [a', b']. Instead of asking the user to determine if all these values correspond to correct matches, we adopt a bisection-like strategy to reduce user interactions: given a list of values, the first question is on the middle, and depending on the answer, the remaining questions will be restricted to either the first or the second half of the list.

3.3.2 Resolving the Uncertainties

From our experiments, we observed several types of common errors: (1) false positive 1:1 mappings due to homonyms; (2) false negative 1:1 mappings; and (3) false negative 1:m mappings. To reduce these errors, this section proposes several methods to determine the uncertainties arising in the matching process and resolve them with user's interactions.

Determine Possible Homonyms: Homonyms are two words pronounced or spelled the same but with different meanings. Here, we use homonyms to refer to two attributes which have very large linguistic similarity but rather small domain similarity. For example, two attributes may both have label type of job, but one is for the duration of jobs (e.g., part time and full time), while the other is for the specialty of jobs (e.g., accountant, clerk, and lawyer). Intuitively, the domain of an attribute reflects its extensional semantics, while the label or name of an attribute conveys its intensional semantics. Two attributes which have highly similar names or labels but very different domains resemble homonyms in linguistics.

Again, we use Figure 3.7 to illustrate homonym attributes and their relationship with the thresholding function. Suppose that cs_1 is the linguistic similarity and cs_2 is the domain similarity. Note that the lower right area of the figure is where potential homonyms could occur, since this is the area where the linguistic similarity is high but the domain similarity is low. Further note that homonym attributes could have a similarity value large enough to be placed above the thresholding line.

To resolve homonym attributes, users are asked to confirm the mappings when the system discovers two attributes with rather high linguistic similarity but very low domain similarity. Since homonym attributes

could potentially confuse the process of learning the threshold, they are resolved first before the learning starts. If two attributes are determined to be homonyms, they will not be considered during the learning process.

Determine Possible Synonyms: Possible synonyms refer to two attributes which could still be semantically similar even if neither their linguistic similarity nor their domain similarity is very high. Possible reasons for the low similarity of two such attributes include: (1) their labels or names do not have any common words; and (2) their domains are actually semantically similar but might not contain a sufficient number of common values so that their domain similarity will be large enough. Examples of such attributes are those positive points located below the thresholding line in Figure 3.7.

To determine potential synonyms, an additional *Check-Ask-Merge* procedure is introduced right after the end of step 2 of the clustering algorithm in Figure 3.2. The procedure is a repeated application of the following steps: (1) *check* if there are two clusters which contain attributes with some common instances, and if yes, choose two such attributes with the largest number of common instances; (2) *ask* the user if two chosen attributes match; and (3) if they match, *merge* the corresponding two clusters.

Determine Possible 1:m Mappings: Although the method described in Section 3.2.3 is quite accurate (as indicated by the experimental results) in identifying 1:m mappings, they may fail to discover some *potential* 1:m mappings.

Intuitively, an attribute e is likely to match with attributes f and g if: (1) the similarity between e and f is very close to the similarity between e and g; (2) f and g are very close to each other on the interface; and (3) e is the only attribute on its interface which satisfies conditions (1) and (2). To reduce the number of questions to the users, we further require that f and g are adjacent to each other on the interface. Note that condition (3) is necessary since otherwise there might be multiple 1:1 mappings instead of one 1:m mapping.

We employ similar rules to find potential 1:m mappings with more than two attributes on the many side. The resolution of uncertain 1:m mappings is carried out at the end of the preliminary-1-m-matching phase when all the automatic methods for identifying 1:m mappings have been applied.

Domain	Leaf Nodes			Internal Nodes			Depth			% of Attrs w/ Inst	Distribution of Simple Domain Types							Comp. Types	
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg		Int	Real	String	Money	Time	Area	CalMon	Date	Other
Airfare	5	15	10.7	1	7	5.1	2	5	3.6	71.9	76	0	26	0	20	0	26	6	9
Automobile	2	10	5.1	1	4	1.7	2	3	2.4	61.4	12	0	37	2	0	0	0	0	1
Book	2	10	5.4	1	2	1.3	2	3	2.3	25.4	0	0	24	0	0	0	0	4	18
Job	3	7	4.6	1	2	1.1	2	3	2.1	70.0	2	0	53	1	0	0	0	0	5
Real Estate	3	14	6.7	1	6	2.4	2	4	2.7	67.8	20	0	39	21	0	12	0	0	7

Table 3.1: Domains and characteristics of the data set

3.4 Empirical Evaluation

We have performed extensive experiments to evaluate the performance of the proposed matching algorithm on discovering both simple and complex mappings among the attributes. This section presents the experimental results.

To isolate the evaluation of schema matching from that of schema extraction, we used manually created schema trees for these experiments. Before the experiments, we also manually identified the mappings and used them as the gold standard to evaluate the matching accuracy.

Data Set: All experiments were also performed on the ICQ data set [3]. As described in Section 2.4, the data set contains query interfaces to Deep Web sources in five domains: airfare, automobile, book, job, and real estate, with 20 query interfaces for each domain.

Table 3.1 shows the details of the data set. For each domain, columns 2–7 show the minimum, maximum, and average number of leaf nodes and internal nodes in the interface schemas of that domain. Columns 8–10 show similar statistics on the depth of the schema trees. (Note that these columns reproduce those in Table 2.2 for the completeness.) Column 9 shows the percentage of the attributes which have instances on their interfaces. The remaining columns show the distribution of simple and composite domain types of the attributes.

Performance Metrics: Similar to [41, 70], we measured the performance of the attribute matching algorithm with three metrics: precision, recall, and F1 [97]. Precision P is the percentage of the mappings identified by the algorithm that are correct; recall R is the percentage of the mappings in the gold standard that are correctly identified by the algorithm; and $F1$ incorporates both precision and recall, computed as $2PR/(R+P)$. Note that a 1:m mapping is counted as m 1:1 mappings, one for each attribute on the many side of the mapping. For example, 1:m mapping $e_i \leftrightarrow (e_j, e_k)$ is counted as two 1:1 mappings: $e_i \leftrightarrow e_j$

Domain	Prec.	Rec.	F1
Airfare	92.0	90.7	91.4
Auto	92.8	92.3	92.6
Book	93.5	92.5	93.0
Job	81.8	83.5	82.6
Real Est.	81.0	96.7	88.1
Average	88.2	91.1	89.5

Table 3.2: The performance of the automatic attribute matching algorithm

Domain	Prec.	Rec.	F1
Airfare	94.1	90.5	92.3
Auto	96.3	91.4	93.8
Book	97.8	92.5	95.1
Job	90.0	71.8	79.9
Real Est.	97.6	93.6	95.6
Average	95.2	88.0	91.3

Table 3.3: The performance with learned thresholds

and $e_i \leftrightarrow e_k$.

Experiments: For each domain, we performed three sets of experiments. First, we measured the accuracy of the automatic attribute matching algorithm. Second, we examined the effectiveness of user interaction in improving the accuracy. Third, we evaluated the contribution of different components.

For all the experiments we conducted, the weight coefficients for the component similarities were set as follows. (1) $\lambda_{ls} = .6$ and $\lambda_{ds} = .4$. This reflects the observation that both the description-level and the instance-level properties of an attribute are very important evidences in identifying the semantics of the attribute, and further that labels are typically more informative than instances. (2) $\lambda_n = 1/6$, $\lambda_l = 3/6$, and $\lambda_{nl} = 2/6$. This reflects the observation that the label of an attribute is more informative than the name of an attribute which may contain acronyms and abbreviations. (3) For the domains whose types convey significant semantic information (such as money and time), we set $\lambda_t = .8$ and $\lambda_v = .2$. For other domains (such as int, real, and string), we set $\lambda_t = 0$ and $\lambda_v = 1$.

3.4.1 The Performance of the Automatic Attribute Matching Algorithm

For all the experiments with the automatic attribute matching algorithm, the clustering threshold was set to zero so that two attributes may potentially be matched as long as they have a positive similarity. Table 3.2 shows, for each domain, the performance of the algorithm measured by precision, recall, and F1. We observe that precisions range from 81% to 93.5%, and recalls range from 83.5% to as high as 96.7% over the five domains. We further observe that on the average, 90% in F1 was achieved. These indicate the effectiveness of the algorithm.

Domain	Prec.	Rec.	F1
Airfare	94.1	90.6	92.3
Auto	96.5	97.6	97.0
Book	98.5	97.1	97.8
Job	95.0	86.4	90.5
Real Est.	95.6	97.6	96.6
Average	96.0	94.0	94.8

Table 3.4: The performance with all user interactions

Domain	Thres.	Hom.	1:m	Syn.	Total
Airfare	4	0/0	1/2	0/0	7
Auto	2	0/1	3/1	0/0	7
Book	2	0/1	5/0	0/1	9
Job	5	3/3	0/3	3/14	31
Real Est.	5	0/2	4/4	2/0	17

Table 3.5: The distribution of different types of questions

3.4.2 Results on User Interaction

In the user interaction experiments, we examined the effectiveness of the proposed methods for interactively learning matching thresholds and resolving uncertain mappings. Results are shown in Tables 3.3–3.5. Table 3.3 shows the matching accuracy with threshold learning only, i.e., the only user interaction was to determine the thresholds. Table 3.4 shows the matching accuracy with both threshold learning and uncertainty resolution.

Table 3.5 shows, for each domain, the number of questions asked for each type of questions. Specifically, the second column shows the number of questions asked to determine the thresholds. The third column shows the number of questions asked to determine homonyms (x/y means that x questions were asked with a "yes" response and y questions were asked with a "no" response). For example, in the book domain, one homonym question was raised but the user responded with a "no" answer. The fourth column shows the number of 1:m questions asked. The fifth column shows the number of questions asked at the end of the clustering process to determine potential 1:1 mappings. The last column shows the total number of questions asked.

Threshold learning: Compare Table 3.3 with Table 3.2, we observe that precisions increase significantly and consistently over all five domains while recalls are all around 90% except for the job domain. The reason for the lower recall in the job domain is that one of the questions happened to be on homonym attributes which have a large similarity value, raising the threshold. Note that a larger threshold typically results in higher precision but lower recall. These observations thus highlight the importance of detecting homonyms before learning thresholds. On the average, the number of questions asked per domain is 3.6, which indicates that our threshold learning approach is very effective.

All user interactions: Compare Table 3.4 with Table 3.3, we observe that recall improves consistently

Domain	None			No 1:m Handling			No Instances			No Tie Res.			All		
	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1	Prec1	Rec	F1	Prec	Rec	F1
Airfare	81.0	66.9	73.3	93.0	81.8	87.0	82.2	83.4	82.8	84.9	87.4	86.1	92.0	90.7	91.4
Automobile	90.1	88.8	89.5	92.7	91.2	92.0	88.9	88.5	88.7	92.8	92.3	92.6	92.8	92.3	92.6
Book	97.7	86.8	91.9	93.5	92.0	92.8	97.7	87.2	92.1	93.5	92.5	93.0	93.5	92.5	93.0
Job	79.1	74.7	76.8	81.6	81.0	81.3	79.7	77.2	78.4	81.8	83.5	82.6	81.8	83.5	82.6
Real Estate	77.8	75.4	76.6	79.8	81.3	80.6	79.6	92.2	85.5	80.1	96.0	87.3	81.0	96.7	88.1
Average	85.1	78.5	81.6	88.1	85.5	86.7	85.6	85.7	85.5	86.6	90.3	88.3	88.2	91.1	89.5

Table 3.6: The contribution of different components

over all domains. In particular, the recall in the job domain increases by nearly 15 percentage points. Detailed results indicate that this was largely due to the resolution of homonyms. In fact, as Table 3.5 indicates, six homonym questions were asked in the job domain and three of them were confirmed by the user. We further observe that the most common type of questions are 1:m mapping questions, and at least one 1:m mapping question was confirmed by the user in all domains except for the job domain. The resolution of potential 1:1 mappings was most effective in the real estate domain. Overall, the total number of questions asked ranges from 7 in both the airfare and automobile domains, to 31 in the job domain.

The overall improvement due to user interaction can be observed by contrasting Table 3.4 with Table 3.2. We note that on the average, precision increases by 7.8 percentage points, recall by 2.9 percentage points, and F1 by 5.3 percentage points. These indicate the effectiveness of user interaction.

3.4.3 Studies on Component Contribution

Table 3.6 shows the contribution of different components in the automatic attribute matching algorithm to the overall performance. We examined the components for handling 1:m mappings, utilizing instances, and resolving ties. To ease the comparison, we reproduce the performance of the complete algorithm (Table 3.4) in the last three columns.

Handling 1:m mappings: Columns 5–7 show the results with the component for handling 1:m mappings removed. By comparing them with the results of the complete algorithm, we can observe that 1:m mappings occur in all the five domains. With the handling of 1:m mappings, recall consistently increases over all domains, with the largest increase (15.4 in percentage point) in the real estate domain. Precision either increases or remains the same in all domains except for the airfare domain. Detailed results indicate that the slight decrease in precision in the airfare domain was due to the relatively worse performance (83.8% in

precision) in finding 1:m mappings for this domain.

Utilizing instances: Since some of the current solutions for matching interface schemas did not utilize instances of attributes, we want to examine the effectiveness of exploiting instances in matching attributes. Columns 8–10 show the results without utilizing the instances. We observe that with the instances, recall increases significantly over all domains, with the largest increase (7.3 percentage points) in the airfare domain. This highlights the importance of instances in improving the matching accuracy.

Resolving ties: Our tie resolution approach was actually motivated by the observation on the interfaces in the airfare domain. The results shown in columns 11–13 indicate that the approach was indeed effective, resulting in an increase of over 7 percentage points in precision and over 3 percentage points in recall in that domain. The improvement can also be observed in the real estate domain.

The aggregate contribution of these components can be observed by contrasting columns 2–4 with the last three columns. As expected, the improvement in recall can be observed over all domains, ranging from 3.5 percentage points in the automobile domain to as high as 23.8 percentage points in the airfare domain. On the average, the recall increases by 12.6 percentage points.

3.5 Summary

This chapter presented an interactive clustering-based algorithm to matching a larger number of interface schemas. The algorithm has been extensively evaluated over varied domains and the results show that it is very effective. The key novelties of our solution include: (1) novel formulation of schema matching as a clustering problem; (2) handling both simple and complex mappings of attributes; (3) incorporating user interaction to learn matching parameters; and (4) interactive resolution of uncertain mappings.

Although our work is done in the context of matching interface schemas, we believe that many of our techniques can be applied to other schema matching tasks. For example, similar clustering-based matching algorithm may be employed to effectively match a large number of relational or XML schemas. Furthermore, our active learning approach for learning matching threshold can be adapted to systematically tune the parameters in other matching systems.

Chapter 4

Web-Assisted Schema Matching

Besides the label mismatch problem addressed in Chapter 3 (Section 3.1), another major challenge to matching interface schemas is the *pervasive lack of data instances*. That is, query interfaces often contain many attributes with no instances at all, such as attributes A_1, B_1, and B_2 in Figure 4.1. Indeed, in the data set used for our experiments (see Section 4.5), the percentage of attributes with no instances ranges from 28.1% to as high as 74.6%. For the attributes that come with instances, the number of such instances is often small and even when the attributes match, their instances are often dissimilar. For example, two attributes A_5 = Airline and B_3 = Carrier match, but the instances of A_5 are mostly North American airliners (e.g., Air Canada) while the instances of B_3 are mostly European airliners (e.g., Aer Lingus).

Matching attributes with no or dissimilar instances is very challenging, since we can only rely on their labels, which are often generic or similar to many other labels. For example, the label of attribute B_1, Departure city, is similar to both the label of A_1 (From city, a matching attribute) and the label of A_2 (Departure date, a non-matching attribute). As another example, we note that two matching attributes A_5 = Airline and B_3 = Carrier have no common words in their labels.

It is important to note that the lack of data instances arises even in traditional schema matching contexts, such as during schema or view integration [83]. However, there the schemas to be matched often contain a variety of *other* meta-data information that can be exploited effectively by current matching techniques [83]. Examples of such meta data include attribute types, cardinality, the structural information among attributes, and semantic integrity constraints. In contrast, by their nature, query interfaces on the Deep Web contain very little or no such meta-data information. Hence, here the lack of data instances severely exacerbates the matching problem. Consequently, it is important to develop solutions that discover data instances for interface attributes, as these solutions can significantly improve the interface matching accuracy.

In this chapter, we describe WebIQ, a solution that learns from both the Surface Web and the Deep Web to automatically discover instances for interface attributes. The solution consists of the following three

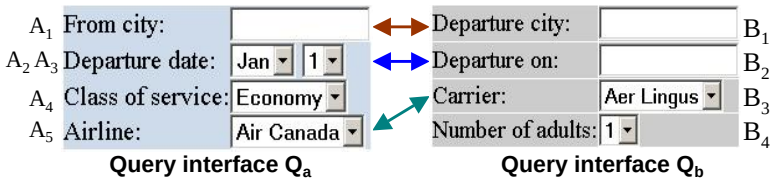


Figure 4.1: Two query interfaces in the air travel domain and semantic matches between them.

PRT Travel...We're going places!
... Other **departure cities such as** Boston, Chicago and LAX available for a surcharge. CHINA - 11 Days Shanghai, Yangtze River Cruise, & Beijing Feb. ...
70428.prttravel.net/fam_news.php - 97k - Supplemental Result - Cached - Similar pages

Figure 4.2: A result snippet from Google.

components:

Discover Instances from the Surface Web: Given an attribute A, such as Departure city, WebIQ formulates *extraction queries* such as "departure cities such as", using attribute label and a set of lexico-syntactic rules [45]. For example, a rule may specify that "if the label L is a singular noun phrase, then form the query '[plural form of L] such as"'.

Next, WebIQ poses the queries to a search engine (e.g., Google) to obtain a set of result snippets. Figure 4.2 shows such a snippet, in response to the above query. WebIQ then examines the snippets to extract candidate instances. For example, from the above snippet, WebIQ will extract three instances: Boston, Chicago, and LAX.

Similar query-the-Surface-Web approaches have also been studied in the AI community, for example to populate ontologies [31]. However, in the context of interface matching, formulating extraction queries is significantly more challenging. This is because attribute labels often take syntactic forms that are not nouns or noun phrases, such as From city (a prepositional phrase). To address this problem, WebIQ performs a *shallow syntactic analysis* on the attribute label, using part-of-speech (POS) tagging [12] and pattern matching, then uses the analysis results to form appropriate queries. It also adds to such queries keywords formed from labels of other attributes, to narrow the scope of the queries.

Since the Web is often noisy, in the next step WebIQ must ensure that the extracted instances are indeed instances of the attribute. Toward this goal, it employs a two-phased validation process. First, in the *outlier detection* phase, WebIQ detects and removes false instances by performing discordancy tests [8], based on a set of type-specific test statistics. Second, in the *Web validation* phase, WebIQ forms a set of validation queries, using attribute label, extracted instance candidates, and a set of validation patterns. For example, a

validation query for instance candidate Boston is "Departure city Boston". WebIQ then poses validation queries to the Surface Web, computes for each instance candidate a validation score, and returns those with sufficiently high scores. This two-phase validation process has an additional advantage that it greatly reduces the number of validation queries posed to search engines.

Borrow Instances from Other Attributes: Given an attribute A, WebIQ also attempts to "borrow" instances for A from other attributes. Specifically, suppose b is an instance of attribute B, then WebIQ will try to ascertain if b can also be an instance of A. Note that this can significantly help us to match A and B. In Figure 4.1, for example, WebIQ can try to ascertain if instance Jan of attribute A_2 = Departure date can also be an instance of attribute B_2 = Departure on, or if instance Aer Lingus of B_3 = Carrier can also be an instance of A_5 = Airline.

To verify that an instance b of attribute B is also an instance of A, one approach is to obtain a set of instances from the Surface Web for A, then check if b is among them. We found that this approach does not work well because b is often not among the top instances for A (as discovered from the Surface Web). Another approach is to form validation queries as described earlier, but using the label of A and the instance b, then check if the validation scores are comparable to those for the (existing) instances of A. We found that this approach does not work well either because the validation scores for b (e.g., Aer Lingus) are often much lower than those for the existing instances of A (e.g., Air Canada).

We observed that a more reliable way to assess the validation scores for b is to compare them with those for the *non-instances* of A (e.g., if A is Airline, then Economy from attribute Class of service is a non-instance of A). The intuition is that the validation scores for the instances and the non-instances of an attribute are likely to be quite separable, and this *separation* can be exploited to accurately classify new instances. Based on this intuition, WebIQ first trains a validation-based classifier for A using instances of other attributes on A's interface as negative examples, then employs the classifier to predict the membership of b.

Validate Borrowed Instances via the Deep Web: Consider again an attribute A. As discussed earlier, if A's label is not in a "benign" syntactic form (e.g., noun or noun phrase), it may be difficult to formulate reliable extraction queries. Furthermore, the extraction queries may fail to obtain any instances from the Surface Web. In these cases, we can borrow instances for A from other attributes, as just discussed. However, it is unlikely that validating them via the Surface Web will work well, given that it is hard to formulate reliable

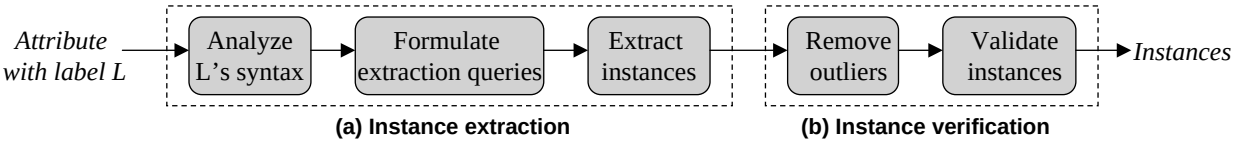


Figure 4.3: Steps in discovering instances from the Surface Web.

extraction queries or that these queries have not returned instances.

To address this problem, WebIQ develops a solution to validate instances via the Deep-Web sources. Specifically, to verify that b is an instance of attribute A, WebIQ submits a query to the data source of A, with A's value set to b, then observes the response from the source. The key intuition is that in many cases the Deep-Web source will be able to distinguish instances of an attribute from non-instances even if the Surface Web cannot. For example, consider an attribute with label from (for the flight origin) on an airfare interface. While both from January and from Chicago might frequently occur on the Surface Web (thus making validating via the Surface Web difficult), often querying the source with the value of attribute from set to Chicago will yield some meaningful results, whereas querying with from = January will not.

In summary, this chapter makes the following contributions:

- A set of novel techniques, as embodied by the WebIQ system, that automatically acquire instances for attributes of query interfaces from the Surface Web and the Deep Web (Sections 4.1-4.3). The techniques also have potential applications in the general schema matching contexts (Section 4.6).
- The incorporation of the above techniques into the schema matcher of IceQ (Section 4.4).
- Extensive experiments over five real-world domains that demonstrate the utility of the proposed techniques. In particular, the results show that acquired instances help improve matching accuracy from 89.5% F1 to 97.5%, at only a modest runtime overhead (Section 4.5).

4.1 Discover Instances from the Surface Web

We now describe the three components of WebIQ in detail. This section describes Surface, the component that discovers instances from the Surface Web, while the next two sections describe the remaining two components. Section 4.4 then discusses how the components are incorporated into IceQ.

Given an attribute A and a constant k, Surface returns up to k instances of A, as gleaned from the Surface Web. It operates in two phases: extraction and verification (Figures 4.3.a-b, respectively).

In the extraction phase, Surface analyzes the syntax of A's label, and formulates a set of extraction queries. It then poses the queries to a search engine, obtains the results, and extracts instance candidates. In the verification phase, Surface first removes statistical outlier candidates, then verifies the rest of the candidates via the Surface Web. Finally, it returns the top k candidates, as ranked by their validation scores (if there are fewer than k candidates, then it returns all of them). The rest of this section describes the two phases in detail.

4.1.1 The Instance Extraction Phase

Analyze Label Syntax: As discussed earlier, an attribute label may take a variety of syntactic forms such as noun phrase (e.g., Departure city and Type of job), prepositional phrase (e.g., From and From city), verb phrase (e.g., Depart from), and even a sentence. Intuitively, it is relatively easier to formulate reliable extraction queries using nouns or noun phrases than other more open-ended forms such as prepositions. As such, this step analyzes an attribute label to extract nouns or noun phrases, which are then used in subsequent steps to form extraction queries.

Specifically, given an attribute A, Surface checks A's label for the occurrence of either a noun phrase, a prepositional phrase (a preposition followed by a noun phrase), or a noun phrase conjunction (a set of noun phrases connected by conjunctives such as "and" and "or", e.g., First name or last name). For a prepositional phrase, the noun phrase after the preposition is obtained. For a noun phrase conjunction, all noun phrases in the conjunction are obtained, and the rest of the instance discovery process is repeated for each noun phrase. If the label does not contain noun phrases, the extraction phase terminates and returns an empty set of instances.

To determine the syntactic form of the label, Surface employs a *shallow syntactic analysis* approach, which involves *part-of-speech (POS) tagging* and *pattern matching*. Specifically, first Brill's tagger [12] is employed to tag the label. The obtained POS tags are then matched against a set of pre-determined patterns to identify the interesting syntactic forms (as described above). For example, the pattern for noun phrases is:

optional determiner + optional modifiers (adjectives/noun-adjectives) + noun + optional post-

Set extraction patterns:
- s1: *Ls such as* NP_1, ..., NP_n
- s2: *such Ls as* NP_1, ..., NP_n
- s3: *Ls including* NP_1, ..., NP_n
- s4: NP_1, ..., NP_n, *and other Ls*

Singleton extraction patterns:
- g1: *the L of the O is* NP
- g2: *the L is* NP
- g3: NP *is the L of the O*
- g4: NP *is the L*

Figure 4.4: Extraction patterns (L: label; Ls: L's plural form; NP: noun phrase; O: object name)

modifier (e.g., prepositional phrase).

Such a pattern matching approach has been shown to be more accurate in many applications than more sophisticated syntactic parsing [66].

Formulate Extraction Queries: Given the noun phrases, this step formulates a set of extraction queries for attribute A. At a high level, we view instance discovery as a *question answering* problem, as commonly understood in AI: we pose a question, the extraction query, to a search engine to obtain a set of instances as the answer. The extraction queries can be regarded as incomplete sentences, and the job of the search engine is to complete the sentences with instances.

Specifically, Surface formulates extraction queries using the noun phrases obtained from the label of A and some domain information from the schema[1] of A. Domain information is used to narrow the scope of formulated queries as much as possible. We consider the following types of domain information:

- The name of the real-world entity that A is associated with (e.g., "book" on a bookstore interface).
- The name of the domain (e.g. "real estate" for a real estate interface), and
- The labels and instances of other attributes in the schema (e.g., "title" and "ISBN" in a bookstore schema).

We note that the name of the object is typically the same as the name of the domain, and further that these information can be obtained automatically.

Extraction queries fall into two categories: *set* extraction queries and *singleton* extraction queries, with the former extracting a set of instances and the latter one instance at a time. The formulation of both types

[1] In the rest of the paper, we use the terms "schema" and "query interface" interchangeably.

of queries is based on a set of *generic* extraction patterns listed in Figure 4.4, where s_i's (g_j's) are the *set* (*singleton*) extraction patterns, respectively. Note that the *set* extraction patterns are similar to those used in [45] for the acquisition of *hyponyms* from natural language texts. Each extraction pattern consists of two parts: *cue phrase* (shown in italic) and *completion* (NP or NP_i's). For example, the cue phrase in s_1 is Ls such as, where Ls is the plural form of the label L, and the completion is a list of noun phrases: NP_1, ..., NP_n, each considered to be an instance candidate for the attribute.

Given the set of extraction patterns, the extraction queries are formed using the cue phrases in the patterns. Specifically, for each pattern, its cue phrase is first *materialized* by replacing L with the noun phrase obtained from the label of attribute A. For example, suppose that A is an attribute in a bookstore schema and has a label author. Then, s_1 will generate authors such as and g_1 will yield the author of the book is. Next, the cue phrases are augmented with the domain information and properly formatted according to the query syntax of search engines, resulting in the final extraction queries. For example, one such extraction query to Google is

"authors such as" +book +title +isbn

where book is the name of the domain, title and ISBN are the labels of some attributes in the schema. Note that double quotes enclose a phrase, while '+' signs request Google to ensure that the results contain the specified keywords.

Extract Instances from the Surface Web: The extraction queries are then posed to a search engine, which is Google in our experiments (using its Web API at www.google.com/apis). For each extraction query, we download top k snippets returned from Google. Finally, we employ a set of *extraction rules* to obtain instance candidates from the snippets. Each rule corresponds to one extraction pattern in Figure 4.4. An extraction rule consists of two parts: the first part identifies the *cue phrase* and the second part extracts the *completion*. For example, the extraction rule for the snippet in Figure 4.2 is: (1) identify the occurrence of the cue phrase "departure cities such as" in the snippet; and (2) extract the list of noun phrases which *immediately* follow the cue phrase, i.e., Boston, Chicago, and LAX.

4.1.2 The Instance Verification Phase

Remove Outliner Instance Candidates: Given a set of instance candidates, Surface prunes the set further in two steps: *pre-processing*, which determines the type of the instance domain and removes candidates

which are not the determined type; and *type-specific detection*, which employs a set of type-specific *test statistics* to detect and remove further outlier candidates.

The pre-processing step employs a set of type-recognizing regular expressions to determine the type of the instance domain. Currently we consider only two types: numeric and string. If the majority of instance candidates (e.g., 80% in our experiment) are either monetary values (e.g., $15,200), integers, or real numbers, the instance domain will be determined to be numeric; otherwise it is string.

Next, the type-specific detection step performs *discordancy tests* [8] with a set of test statistics, all assumed to be normally distributed. An instance candidate is considered to be an outlier if its test statistic is at least three standard deviations away from the average over all the candidates.

For instances of numeric type, the test statistics are their values. For example, it is unusual for the price of a book to be $10,000. For instances of string type, the test statistics are:

- the number of words in the instance, e.g., it is unusual for a person's name to have more than four words;
- the number of capital letters in the instance, e.g., the first letter of a city name is typically capitalized;
- the length of the instance (i.e., the number of characters in the instance), e.g., it is unusual for the make of a vehicle (such as Honda, Toyota) to have over 20 characters; and
- the percentage of numerical characters in the instance, e.g., the ISBN of a book typically has ten digits and no more than three hyphens or white spaces.

Validate Instances via Surface Web: Web validation further removes false instances from the candidate set by assessing the *semantic* connection between the candidates and the attribute, based on their co-occurrence statistics on the Surface Web. The idea is that the meaning of an instance x can be partly characterized by the contexts where x appears. As such, if x is indeed an instance of attribute A, we expect that the label of A may frequently co-occur with x. Such *co-occurrence* statistics can then be exploited to measure the semantic connection between A and x.

As an example, suppose that A has label make (for automobiles). Consider Honda, one of A's instances. We would expect that make can often be found in the context of Honda in varied ways over the Surface Web pages, e.g., "a variety of makes such as Honda, Mitsubishi", "Make: Honda, Model: Accord", and "This car's make is Honda", as indicated by Google.

Based on the above observation, for each instance candidate x of attribute A, we form several *validation queries* using a set of validation patterns. Each validation pattern has two parts: a validation phrase V and the candidate x. Currently, we consider two types of validation patterns:

- *Proximity-based* pattern "L x", where $V = L$, the label of A. This pattern simply considers the proximity of L and x. For example, this pattern gives "make honda" as the validation query for L = make and x = Honda.

- *Cue phrase-based* patterns such as Ls such as x and such Ls as x, which utilize the cue phrases in the extraction patterns (Figure 4.4) as the validation phrases. For example, makes such as Honda is a validation query formed by these patterns.

Intuitively, validation phrases serve the purpose of distinguishing instances of an attribute from *non-instances*. In other words, we expect that instances of an attribute tend to occur more frequently with the validation phrases than non-instances. A possible measure on the co-occurrence of an instance with a validation phrase is the number of hits obtained from a search engine for the validation queries constructed as above. A problem with this measure is the potential bias towards popular instances (or non-instances).

To handle this problem, we adapt the *pointwise mutual information* (PMI) [31] to measure the co-occurrence. Specifically, consider a validation phrase V and an instance candidate x. Let $V + x$ be the validation query (which combines V and x). The PMI between V and x, denoted as $\mathrm{PMI}(V, x)$, is then given by:

$$\frac{\mathrm{NumHits}(V + x)}{\mathrm{NumHits}(V) * \mathrm{NumHits}(x)},$$

where $\mathrm{NumHits}(V)$ and $\mathrm{NumHits}(x)$ are respectively the number of hits obtained from a search engine on the validation phrase and the instance candidate, and $\mathrm{NumHits}(V + x)$ is the number of hits on the validation query. Intuitively, PMI between V and x measures the statistical dependence of V and x such that a larger PMI indicates a stronger dependence.

Denote the set of validation phrases for attribute A as $\mathcal{V} = \{V_1, V_2, ..., V_n\}$. The confidence score of x being an instance of A is then taken to be the average PMI score of x, i.e., $\sum_i(\mathrm{PMI}(V_i, x))/n$. Surface then returns the k instance candidates with top score.

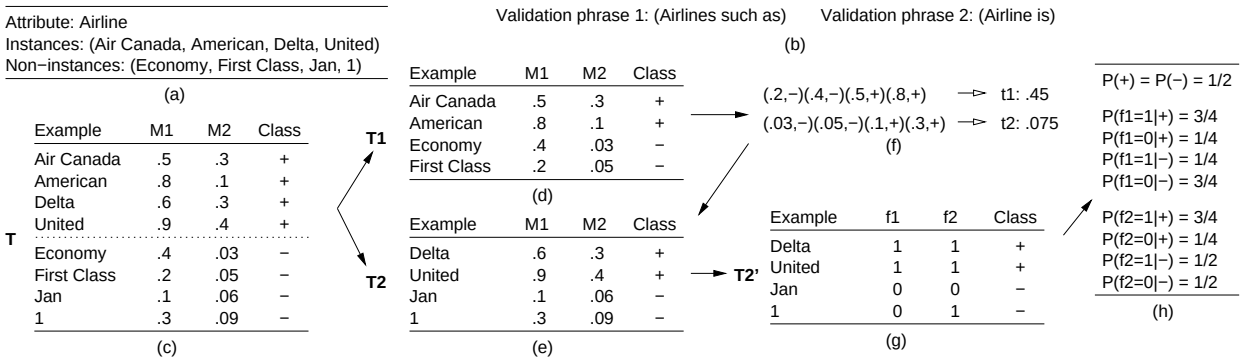


Figure 4.5: An example on training the validation-based classifier.

4.2 Borrow Instances from Other Attributes

Given an attribute A, WebIQ can also "borrow" instances for A from other attributes. Specifically, suppose b is an instance of attribute B, then WebIQ will try to verify if b can also be an instance of A. This verification process can be done via the Surface Web or the Deep Web. This section describes Attr-Surface, the WebIQ component that verifies instances via the Surface Web. The next section describes Attr-Deep, the component that verifies instances via the Deep Web.

To verify if instances of B can be instances of A, Attr-Surface first learns an *instance classifier* for A from a training set, then employs the learned classifier to classify the instances of B. Many previous works on schema matching [25, 83] have utilized varied forms of instance classifiers, but they all rely on a large number of training examples. Such a large training set is not available from the interfaces, since an interface attribute typically has only a handful of instances available on its interface. Furthermore, it might be expensive to obtain a large number of instances from the Web. To address these challenges, we develop a novel approach to learning an instance classifier for an interface attribute. The learned classifier can be regarded as a variant of traditional naive Bayes classifier, but based on a validation scheme. Another distinct aspect of the approach is that the training of the classifier is fully *automatic*, with no needs for manually prepared training examples. We now describe the classifier and its training algorithm in detail.

4.2.1 A Validation-based Naive Bayes Classifier

A naive Bayes classifier [74] is a probabilistic function which, given an object represented by a feature-value vector and a finite set of classes, predicts the class membership of the object. The prediction is based on prior probabilities of the classes, class-conditional probabilities of the object, and the assumption that the

features of an object are independent of each other given its class label.

More precisely, consider an object x represented by a vector $<f_1, f_2, ..., f_n>$, where f_i is the value of the i-th feature of x. Assume two classes: c and $\neg c$. The probability of x belonging to the class c, denoted by $P(c|x)$, is then given by:

$$\frac{P(c)\Pi_i P(f_i|c)}{P(c)\Pi_i P(f_i|c) + P(\neg c)\Pi_i P(f_i|\neg c)}. \tag{4.1}$$

Clearly, features for a class should capture the *salient aspects* of the instances of the class so that they can be distinguished from the non-instances of the class. Our key observation is that the statistics obtained from the Surface Web on the validation queries for an attribute can be utilized as the features for the attribute. Specifically, we expect that the PMI scores of validation queries for instances of an attribute are likely to be much higher than those for non-instances of the attribute, and that this distinction can be exploited to perform classification.

Motivated by the above observation, we represent an object by its thresholded validation scores. Specifically, consider attribute A and an object x. Let $\mathcal{V} = \{V_1, ..., V_n\}$ be the set of validation phrases associated with A. First, we obtain x's validation scores and store them in a *validation vector* $M = <m_1, ..., m_n>$, where m_i is x's validation score on the i-th validation query of A. Let t_i be the threshold for the i-th validation score (which we show how to estimate in the next subsection). Then we use the t_i's to represent x with an n-dimensional vector $<f_1, ..., f_n>$, where $f_i = 1$ if $m_i > t_i$, and 0 otherwise. Intuitively, the thresholds characterize the separation in validation scores between the instances and the non-instances of A.

4.2.2 The Training Algorithm

Training the classifier for an attribute A amounts to estimating the probabilities in Formula 4.1. The training process can be divided into three steps: training set preparation, threshold estimation, and probability estimation. We now describe each step in detail. Figure 4.5 illustrates the process of training the classifier for the Airline attribute (A_5) on the interface Q_a shown in Figure 4.1.

1. Create Training Set T: First, we obtain a set of instances and non-instances for A. The non-instances of A are obtained from *other* attributes on the *same* interface as A. For example, Figure 4.5.a shows instances and non-instances of Airline used for the training. Next, for each instance of A, we obtain its validation scores as described in Section 4.1.2, using the Surface Web, and then turn it into a positive example. Similarly, we create negative examples using non-instances of A. For example, Figure 4.5.b

shows two validation phrases associated with the attribute A_5, and Figure 4.5.c shows the obtained training set T, where the m_1 and m_2 columns show the first and second validation scores, respectively.

Finally, T is divided into two parts: T_1 and T_2, where T_1 is used to estimate the thresholds and T_2 to estimate the probabilities. For example, Figure 4.5.d and 4.5.e shows T_1 and T_2, respectively. Note that T_1 contains the first two positive examples and the first two negative examples in T.

2. Estimate the Thresholds: In this step, we use T_1 to estimate thresholds t_i's. Consider threshold t_i for the feature f_i. Intuitively, a good threshold should be the one that best separates positive and negative training examples in T_1. For this, we use *information gain* [74] to measure the quality of t_i. Specifically, suppose that t_i divides T_1 into T_{11} (where $f_i < t_i$) and T_{12} (where $f_i \geq t_i$). The information gain with respect to t_i is then computed as $E(T_1) - (|T_{11}|/|T_1| * E(T_{11}) + |T_{12}|/|T_1| * E(T_{12}))$, where $E(x)$ denotes the entropy of x. In other words, we choose t_i such that it leads to the largest reduction in the entropy of the training examples in T_1. For example, Figure 4.5.f shows the derivation of t_1 and t_2.

3. Estimate the Probabilities: In this step, we first apply the learned thresholds t_i's on T_2 to transform each validation vector into a feature vector. This results in T_2' as shown in Figure 4.5.g. T_2' is then used to estimate the probabilities. Specifically, $P(f_i = 1|+)$ is estimated to be the percentage of positive examples in T_2' with f_i = 1. To avoid extreme 0/1 probability estimates, Laplacean smoothing [74] is applied. Other conditional probabilities are estimated similarly. Figure 4.5.h shows the estimated probabilities with the smoothing (e.g., $P(f_1 = 1|+) = (2 + 1)/(2 + 2) = 3/4$).

4.3 Validate Instances via the Deep Web

Besides validating instances via the Surface Web, as described in the previous section, WebIQ can also validate instances via the Deep Web. It implements this validation scheme in a third component called Attr-Deep. Given an attribute A and a borrowed instance x (of attribute B), Attr-Deep proceeds as follows.

Formulate and Submit a Query: First, a probing query is formulated by setting the value of A to x and the values of other attributes to their default values. Note that the query interface may contain some attributes that do not have instances. The default values for these attributes are typically empty strings. (Our experiments indicate that many interfaces permit partial queries where the values of some attributes can be left unspecified.) Next, the probing query is posed to the source.

Analyze the Response: This step applies several heuristics to analyze the response page from the source and determine if the submission was successful. We employ a variant of the heuristics used for a similar purpose in [82].

To reduce the number of queries to the source, if the submission is successful for at least one third of the instances of B, then we assume that all instances of B are instances of A.

4.4 Incorporating WebIQ into the Schema Matcher

We now describe how to incorporate the above three components of WebIQ into the schema matcher in IceQ. The incorporation proceeds in two steps:

Instance Acquisition: Let $\{X_1, X_2, \ldots, X_n\}$ be the set of all attributes over all query interfaces. This step employs WebIQ to gather instances for these attributes. Consider attribute X_1. WebIQ gathers instances for X_1 as follows:

1. If X_1 has no instances, then gather instances for X_1 via the Surface Web, using the Surface component (Section 4.1).

   (a) If this gathering is successful, that is, at least k instances have been gathered, for a pre-defined k, then stop.

   (b) Otherwise, borrow instances for X_1 from $X_2, \ldots, X_n$ and validate them via the Deep Web, using the Attr-Deep component (Section 4.3). The reason that WebIQ does not validate them via the Surface Web is because it is unlikely to be successful, given that the instance gathering via the Surface Web in Step 1.a has been unsuccessful.

2. If X_1 has several pre-defined instances, then borrow instances for X_1 from $X_2, \ldots, X_n$ and validate them via the Surface Web, using the Attr-Surface component (Section 4.2). The instances cannot be validated via the Deep Web because X_1 accepts only *pre-defined* values. Thus, we cannot set the value of X_1 on the query interface to a borrowed value, if that value is not in the set of pre-defined values for X_1.

   Note that we can also obtain instances for X_1 via instance discovery on the Surface Web. However, we do not consider that possibility in the current scheme, to minimize the overhead caused by querying

the search engine.

In Steps 1.b and 2, to minimize overhead, WebIQ does not borrow instances from *all* attributes. Instead, it borrows only from those attributes whose domains are deemed potentially similar to that of X_1. Specifically, consider an attribute X_i ($i \neq 1$) from a different interface as X_1. There are two cases: (1) X_1 does not have any pre-defined values (as in Step 1.b). In this case, the domain of X_i is likely to be similar to that of X_1 if the labels of X_1 and X_i are similar (so they are likely to match) and the domain of X_i is very different from the domain of any other attribute Y on the same interface as X_1 (intuitively, if Y and X_1 have similar domains, it is very unlikely that Y has some pre-defined values while X_1 does not). (2) X_1 has a set of pre-defined values (as in Step 2). In this case, the domain of X_i is likely to be similar to that of X_1 if there are at least two values, one from each domain, which are very similar.

The above steps are then repeated to gather instances for X_2, X_3, and so on.

Schema Matching: Once WebIQ has gathered instances for all attributes $X_1, X_2, \ldots, X_n$, the schema matching algorithm of IceQ as described in Chapter 3 is employed as usual to match these attributes. Recall that IceQ employs an interactive clustering-based matching algorithm which group attributes into clusters, each of which contains a set of matching attributes. To cluster, given any two attributes A and B, IceQ computes a similarity score based on the similarity of their labels and instances. Therefore this computation can benefit significantly from additional instances gathered by WebIQ, as our experiments in Section 4.5 will show.

4.5 Empirical Evaluation

This section presents experimental results on evaluating WebIQ. The goal of these experiments was to examine the impact of Web assistance via WebIQ on the matching accuracy. All experiments used the ICQ data set [3]. This data set was also used in the schema matching experiments in Chapter 3. Recall that the data set contains query interfaces in five real-world domains – airfare, automobile, book, job, and real estate, with 20 query interfaces in each domain.

The first five columns of Table 4.1 show the characteristics of the data set. For each domain, it shows the average number of attributes per interface (Column 2), the percentage of interfaces which contain at least one attribute with no instances (Column 3), and among these interfaces, the percentage of attributes without

Domain	#Attr	IntNoInst (%)	AttrNoInst (%)	ExpInst (%)	Instance Acquisition	
					Surface (%)	Surface+ Deep (%)
Airfare	10.7	85	32.2	100	19.0	81.1
Auto	5.1	95	28.1	100	58.7	82.2
Book	5.4	85	38.6	98	84.4	84.4
Job	4.6	100	74.6	83.1	72.2	72.2
Real Est	6.5	95	30.0	66.7	49.1	56.3
Average	6.7	92	40.7	89.6	56.7	75.2

Table 4.1: Characteristics of our data sets and results on gathering instances

instances (Column 4). We can observe that on the average, 92% of the interfaces contain attributes with no instances, and 40.7% of the attributes on these interfaces have no instances. These indicate that the lack of instances is *pervasive*.

Further, we manually examined the attributes with no instances to see if it is reasonable to expect that their instances can be found on the Surface Web, taking into considerations the observation that it is difficult to obtain instances for *generic* attributes (e.g., keyword and description) and attributes related to personal information (e.g., buyer id and reference number). The result is shown in Column 5 of Table 4.1. We observe that on the average, we expect to be able to obtain instances for nearly 90% of the attributes. This suggests the potentials of a WebIQ-like approach.

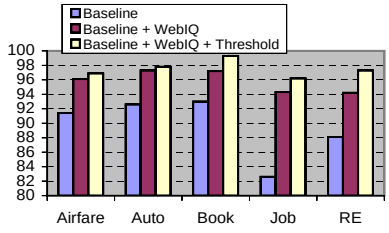


Figure 4.6: Matching accuracy

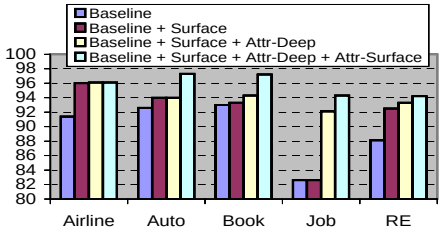


Figure 4.7: Component contributions

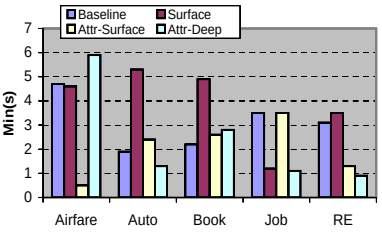


Figure 4.8: Overhead analysis

Instance Acquisition: In the first set of experiments, we evaluated the effectiveness of WebIQ on instance acquisition, focusing on the attributes with no instances. For each such attribute, if WebIQ obtains at least 10 instances, then the acquisition process is determined to be successful.

The results are shown in the last two columns of Table 4.1. Column 6 shows the success rates when we only employed the Surface component of WebIQ (Section 4.1). Column 7 shows the success rates when

we also used its Attr-Deep component (Section 4.3).

Acquisition via the Surface Web: We observe that with the Surface component, the success rate ranges from 19% in the airfare domain to 84.4% in the book domain, with an average of 56.7%. Detailed results reveal the following observations.

First, the relatively low success rate in the airfare domain is largely due to the fact that the labels of many attributes with no instances are either prepositions or verb phrases, e.g., from and depart from. As discussed earlier, it is very challenging to form reliable extraction queries for these attributes. Second, several attributes in the auto domain have very ambiguous labels, e.g., zip for "zip code", which greatly reduces the success rate in that domain. Third, the real estate domain has many attributes for measurement units (e.g., square feet and acreage). We observe that for these attributes, the extraction patterns are not as effective. Finally, we note that both the book and job domains have very high success rates. This is not surprising, given that the labels of most attributes with no instances in these domains are either nouns or noun phrases such as publisher, author, company name, and city. The extraction patterns tend to be very effective for these attributes.

Instance Validation via the Deep Web: We observe that the additional Attr-Deep component significantly increases the success rate in both the airfare and auto domains. Interestingly, these are the domains for which the Surface component had difficulties in getting instances from the Web. On the average, the success rate increases by 18.5%. These indicate the effectiveness of the Attr-Deep component.

Interface Matching with WebIQ: In the second set of experiments, we evaluated the extent to which WebIQ helps improve the matching accuracy of IceQ (see Section 4.4). Recall from Section 3.4 that we measure the matching accuracy via three metrics: precision, recall, and F1.

For each domain, we performed two experiments. First, we collected results of both IceQ and IceQ + WebIQ with no thresholding. That is, the clustering threshold of IceQ is set to zero so that as long as two attributes have a positive similarity, they may *potentially* be matched. Then, the results of IceQ + WebIQ were recollected with the threshold τ uniformly set to .1, which is about the average of the thresholds learned for the five domains (see also Section 3.4).

Figure 4.6 shows the results. For each domain, it shows three bars which represent, from left to right, the accuracy (in F1) of IceQ, IceQ + WebIQ, and IceQ + WebIQ with thresholding (in the figure, IceQ is referred to as "baseline").

The results show that IceQ + WebIQ significantly improved the accuracy over IceQ, across all five domains. The improvement ranges from 4.2% in the book domain to 11.7% in the job domain. On the average, the accuracy increases from 89.5% to 95.8%. The thresholding further increases the accuracy to 97.5%. Detailed results indicate that the thresholding significantly improved the precision while maintaining the recall. These indicate the effectiveness of WebIQ.

Component Contributions: In the third set of experiments, we examined the contribution of individual components in WebIQ to the overall accuracy. Figure 4.7 shows the results. For each domain, it shows four bars which respectively represent the accuracy of IceQ and IceQ with WebIQ's components consecutively incorporated.

The results show that Surface significantly improved the matching accuracy. For example, it resulted in a 4.6% increase in the airfare domain and a 4.4% increase in the real estate domain. Attr-Deep had the most significant impact (a 9.5% increase) in the job domain. Finally, Attr-Surface was very effective in four of the five domains, and on the average, it improved the accuracy by 1.8%.

Overhead Analysis: In the last set of experiments, we examined the overhead incurred by WebIQ. Results are shown in Figure 4.8. For each domain, it shows the times (in minutes) which IceQ + WebIQ spent in matching attributes (the first bar), gathering instances from the Web (the second bar), validating instances via the Surface Web (the third bar), and validating instances via the Deep Web (the last bar). In other words, the last three bars show the overhead incurred by each individual component of WebIQ.

We can observe that the matching time ranges from 1.9 minutes in the auto domain to 4.7 minutes in the airfare domain. Time spent by Surface ranges from 1.2 minutes in the job domain to 5.3 minutes in the auto domain. This time varies in different domains due to different numbers of queries sent to Google. For example, the total number of extraction and validation queries for the job domain is 432 (over 20 source schemas). Note that the typical retrieval time from Google for one query is 0.1–0.5 second.

Time spent by Attr-Surface was at most 3.5 minutes (in the job domain), and time spent by Attr-Deep was at most 5.9 minutes (in the airfare domain). The total overhead ranges from 5.7 minutes in the real estate domain to 11 minutes in the airfare domain. These results indicate that it is possible to employ WebIQ without incurring a significant overhead.

4.6 Summary

In this chapter, we described a set of novel techniques, as embodied by the WebIQ system, that automatically acquire instances for attributes of query interfaces from the Surface Web and the Deep Web. We showed how the techniques can be incorporated into IceQ to improve the matching accuracy. Extensive experiments over varied real-world domains demonstrate the utility of our approach. In particular, the results show that acquired instances help improve the matching accuracy from 89.5% (F1) to 97.5%, at only a modest runtime overhead.

Besides improving the effectiveness of the current solutions, our future work will study how to transfer our techniques to other contexts, such as mining the extensive bioinformatics literature to help match schemas of data sources in that domain, and mining text documents that accompany real-world database schemas for further metadata information. Overall, we believe the incorporation of shallow natural language processing techniques over corpora of domain data can greatly help semantic integration tasks, including matching Deep Web query interfaces, schema matching, and record linkage. Our current work is a first step in this direction.

Chapter 5

Schema Merging

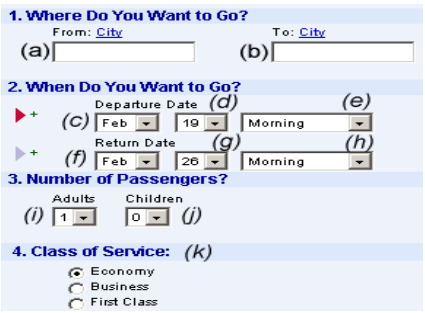
In Chapters 3 and 4, we described the proposed schema matching solutions. In this chapter, we present LMax, a novel schema merger. Given a set of interface schemas, LMax creates a *unified* schema based on the discovered mappings of interface attributes. Such a unified schema should encompass all *unique* attributes over the given set of interfaces and should be structurally and semantically well-formed.

To illustrate, Figure 5.1.a shows a query interface to an airfare source and Figure 5.1.b shows its ordered-tree schema (see also Section 2.1). Furthermore, Figure 5.1.c shows the schema S_v for the interface to a different airfare source, where attribute marked with x' matches with attribute x in schema S_u of Figure 5.1.b. Figure 5.1.d shows a unified schema which integrates schemas S_u and S_v. Note that such a unified schema will permit users to formulate queries not only on attributes common to both schemas, but also on attributes available only in one of the schemas, e.g., time in S_u and airline in S_v.

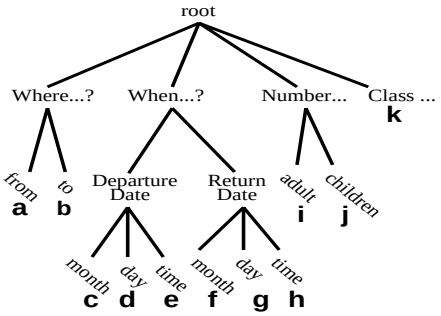
Compared to schema matching [41, 43, 107, 98], the problem of merging schemas has received relatively little attention. As in schema matching, the fundamental challenges to the merging of interface schemas arise from the *scale* of the problem and the *diversity* of sources. Source interfaces are greatly diversified due to the autonomous nature of the sources. As a result, *structural conflicts*, as exemplified by the fact that different interfaces may represent a different set of attributes and may organize the attributes in quite different ways, are prevalent over interface schemas. For example, while schema S_u arranges the attributes by location and date, schema S_v groups them on departure and return.

Resolution of structural conflicts over a large scale of interface schemas presents serious challenges that call for a novel, *automated* solution. Traditional approaches to schema merging [9] are typically either manual or relying on a set of resolution rules prescribed by domain experts. As a result, it is difficult for such approaches to scale up to a large number of diversified schemas.

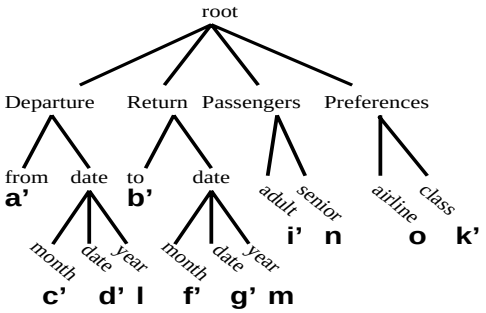
In this chapter, we present a first, *systematic* study on the problem of merging a *large scale* of interface schemas. We start by proposing a novel formulation which views schema integration as an *optimization*



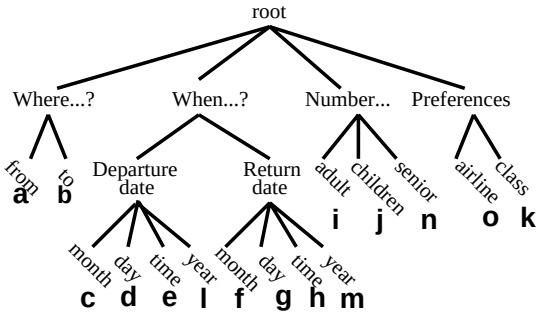
(a) An airfare query interface Q



(b) S_u: the schema of Q



(c) S_v: the schema of a different airfare interface



(d) A schema unifying S_u and S_v

Figure 5.1: Examples of query interface, schema, and unified schema

problem, where each interface schema in essence expresses certain preferences over how the unified schema should look like, and the goal of schema merging is to construct a unified schema such that these preferences are maximally satisfied. This optimization framework thus represents a novel alternative to the traditional pairwise merging approach to schema integration, and provides a principled foundation for evaluating the quality of integrated schemas.

The preferences from interface schemas can be formally represented by two types of constraints: (a) *structural constraints*, which are expressed on the lowest common ancestor (LCA) of attributes and restrict the structure of the unified schema; and (b) *precedence constraints*, which are based on the sequence in which attributes appear on the interface and restrict the ordering of attributes and attribute groups in the unified schema.

Since the optimization problem can be shown to be NP-complete, we propose a novel approximation algorithm LMax, based on recursive applications of *clustering aggregation* [33]. The key idea is to view the construction of the unified schema as a process of forming recursive partitions over a set of attributes.

Merging interface schemas faces another major challenge. As discussed in Section 2.4.4, the structure

of some query interfaces may be implicit, which poses challenges to the schema extractor. As a result, the obtained schema may be *irregular* in the sense that it may not sufficiently reflect the grouping relationships among the attributes on the interface. To address this challenge, a key observation is that the irregularities in these schemas may be identified with the help of other schemas, assuming that at least some schemas in the input are regular. We exploit this observation to propose an extension to LMax and show that it greatly boosts LMax's performance.

We further extend LMax to produced an *ordered* schema. We describe an algorithm which orders the elements in the unified schema such that the precedence constraints from the input schemas are satisfied as much as possible. This algorithm can be regarded as a variant of the topological sorting algorithm.

In summary, this chapter makes the following contributions:

- A *first*, systematic study on merging a *large scale* of interface schemas.
- A *novel* formulation of schema merging as an optimization problem (Section 5.1).
- An effective merging algorithm based on clustering aggregation (Sections 5.2 & 5.3).
- An ordering algorithm for producing ordered schemas (Section 5.4).
- *Extensive* experiments over varied real-world domains. (Section 5.5).

5.1 Schema Merging as an Optimization Problem

This section first reviews the definition of interface schema, then formally defines constraints and the schema merging problem.

Recall from Section 2.1, we model query interface with an *ordered tree* schema, where leaf and internal elements respectively correspond to attributes and attribute groups on the interface. For example, Figure 5.1.b shows the schema S_u for the interface in Figure 5.1.a. Note that to ease the presentation, the attributes on the interface are numbered (from a to k). In the following, we will also use parenthesis notation to represent schemas. For example, schema S_u may be represented as ((a, b) ((c, d, e) (f, g, h)) (i, j) k).

As discussed earlier, each source schema essentially expresses certain preferences on the unified schema. These preferences can be encoded in two types of constraints: (a) *structural constraints*, which are expressed on the ancestor-descendant relationship of the lowest common ancestor (LCA) of attributes, and

thus restrict the structure of the unified schema; (b) *precedence constraints*, which are expressed on the ordering of attributes, and thus restrict the order of elements in the unified schema. We now formally define these constraints. Note that structure constraints are similar to the LCA constraints used to measure the performance of the schema extraction algorithm (Section 2.4).

Definition 9 (**Structural constraint**) Consider a schema S and denote the lowest common ancestor of two attributes x and y in S as LCA(x, y). Consider three attributes x, y and z in S. We say that there exists a *structural constraint* in form of $(x, y)z$ from S, if $\mathrm{LCA}(x, y) < \mathrm{LCA}(x, z)$ and $\mathrm{LCA}(x, y) < \mathrm{LCA}(y, z)$, where $n_1 < n_2$ denotes that element n_1 is a proper descendant of element n_2. □

Example 16 $(a, b)c$ is a structural constraint from the schema S_u. Intuitively, it indicates, according to S_u, attributes a and b (both on the location of flight) are more closely related than either to c (which is on the date of flight). □

Given a set of structural constraints without conflicts, [5] gives a polynomial-time algorithm to construct a tree which satisfies all the given constraints. But when there are conflicts in the given structural constraints, the algorithm in [5] can not be applied. In our integration problem, conflicts are prevalent. For example, $(a', c')b'$ from S_v conflicts with $(a, b)c$ from S_u. In general, constraint $(x, y)z$ conflicts with $(x, z)y$ and $(y, z)x$. Given two conflicting constraints, it is impossible to find a unified schema which satisfies both constraints.

Definition 10 (**Precedence constraint**) Consider an ordered-tree schema S and a sequence of attributes, denoted as q_s, obtained from a *pre-order* traversal of S. We say that there exists a precedence constraint between two attributes x and y, denoted as $x \prec y$, from the schema S, if x appears *before* y in q_s. □

Example 17 The sequence q_{s_u} for the schema S_u is $<a, b, c, d, e, f, g, h, i, j, k>$. As such, $a \prec b$ and $a \prec d$ are two precedence constraints from S_u. □

Based on the above definitions, we cast schema integration as an optimization problem:

Integration problem OPT**:** Given a set of interface schemas $\mathcal{S}$ with a set of *distinct* attributes A, find a unified schema G such that (1) G's leaf elements are attributes in A; (2) the number of *structural constraints* from the schemas in $\mathcal{S}$ that are satisfied by G is maximized; and (3) the number of *precedence constraints* from the schemas in $\mathcal{S}$ that are satisfied by G is maximized.

Proposition 1 OPT *is NP-complete.* □

Proof (sketch): NP-completeness of OPT can be proved by utilizing the results on the NP-completeness of its two sub-problems. First, it can be shown that the sub-problem of OPT with conditions (1) and (2) is NP-complete by a reduction from *3-partite perfect matching* [76]. Second, it be shown that the sub-problem of OPT with conditions (1) and (3) is NP-complete by a reduction from *cyclic-ordering* [20]. The NP-completeness of OPT then follows by a reduction from either of its sub-problems.

5.2 Approximating OPT via Clustering Aggregation

This section presents the algorithm LMax which gives an approximate solution to OPT. Essentially, LMax views the construction of a unified schema as a process of forming recursive partitions over a set of attributes such that *structural constraints* from the interface schemas are satisfied as much as possible. We will extend LMax to produce ordered schema in Section 5.4. Before we describe the algorithm in detail, we first give an example on LMax.

Example 18 Consider integrating schemas S_u and S_v. To start with, LMax is given a set A of 15 unique attributes (numbered a–o) over two schemas. Then, in the first iteration, LMax creates a root node r and forms a partition over the attributes in A such that it agrees with the partitions given by interface schemas as much as possible. The partition given by an interface schema S on the attributes in A is obtained by first *restricting* (more precisely defined in Section 5.2) S on A and then extracting the top level clusters in the reduced schema. For example, $\{a, c, d, l\}$, $\{b, f, g, m\}$, $\{i, n\}$, and $\{o, k\}$ are the top level clusters of schema S_v.

Next, LMax creates a list of children for r, one for each cluster in the partition. The same process is then recursively applied to each child with the attributes in its associated cluster. □

We now introduce several necessary concepts.

Definition 11 (**Cluster, maximum cluster, and clustering**) Consider a schema S which contains a set of attributes A. For each node n in S, we define a *cluster* C_n as the set of attributes (i.e. leaf elements) in the sub-tree rooted at n. A cluster C is a *proper* cluster if $C \subset A$. A proper cluster is a *maximum cluster* if it is not a subset of any other proper clusters in S. The set of all maximum clusters in S forms a *clustering* over the attributes in A. □

LMAX($\mathcal{S}$, A, r)
Input: $\mathcal{S}$, a set of interface schemas;
A, a set of distinct attributes over the schemas in $\mathcal{S}$.
Output: r, the root of a unified schema G

---

1. if $A = \{a\}$ /* A has only one attribute */
   $r \leftarrow$ NODE(a)
2. else if $A = \{a, b\}$
   $r_1 \leftarrow$ NODE(a), $r_2 \leftarrow$ NODE(b), $r \leftarrow$ NODE(r_1, r_2)
3. else /* A has at least three attributes */
   a. $\{C_{r_1}, C_{r_2}, ..., C_{r_k}\} \leftarrow$ PARTITION(A, S)
   b. **for** each C_{r_i}, $1 \leq i \leq k$, **do**
      $\mathcal{S}' \leftarrow \mathcal{S}|C_{r_i}$
      LMAX($\mathcal{S}'$, C_{r_i}, r_i)
   c. $r \leftarrow$ NODE(r_1, ..., r_n)

Figure 5.2: The LMax algorithm

Example 19 The maximum clusters in S_u are $\{a, b\}$, $\{c, d, e, f, g, h\}$, $\{i, j\}$, and $\{k\}$, each associated with a child of the root. □

Definition 12 (**Restriction**) A *restriction* of a schema S on a set of attributes X, denoted as $S|X$, is a schema given by: (1) pruning all attributes in S which are not in X; (2) pruning the internal node if all its children are pruned; (3) replacing the internal node having only one child with its child.

A *restriction* of a set of schemas $\mathcal{S} = \{S_1, ..., S_n\}$ on X, denoted as $\mathcal{S}|X$, is a set of schemas $\{S_1|X, ..., S_n|X\}$. □

Example 20 $S_u|\{a, b, c, d, k\}$ = ((*a*, *b*) (*c*, *d*) *k*). □

Based on the above definitions, LMax algorithm is given in Figure 5.2. Note that NODE(a) creates a leaf node with attribute a; and NODE(n_1, ..., n_k) creates an internal node with n_i's as the children.

Given a set of schemas $\mathcal{S} = \{S_1, S_2, ..., S_n\}$ and a set of unique attributes A over the schemas in $\mathcal{S}$, LMax builds a unified schema G as follows. If A has only one attribute a, it simply returns NODE(a) as the root of G. If A has only two attributes a and b, G is a tree with two leaves: NODE(a) and NODE(b). Otherwise, it first forms a partition $P = \{C_{r_1}, C_{r_2}, ..., C_{r_k}\}$ over the attributes in A. Then for each cluster C_{r_i}, it recursively creates a sub-tree rooted at r_i based on a set of restricted schemas $\mathcal{S}|C_{r_i}$. Finally, it returns NODE(r_1, r_2, ..., r_k) as the root of G.

We now describe the PARTITION function (step 3.a), the key component of the LMax algorithm, in detail.

PARTITION: PARTITION(A, $\mathcal{S}$) finds a partition over the attributes in A such that the structural constraints from the schemas in $\mathcal{S}$ are satisfied as much as possible.

Consider a partition P over A where P = $\{C_1, C_2, ..., C_k\}$. Consider further a schema $S \in \mathcal{S}$. Suppose that M = $\{C'_1, C'_2, ..., C'_l\}$ is a set of the maximum clusters in S. We observe that, to satisfy as many structural constraints from S as possible, P should be such that if two attributes x and y are in the same cluster $C_i \in M$, then both x and y should also be placed in the same cluster, say $C_j \in P$. Otherwise, all constraints of the form $(x, y)z$ will be violated. On the other hand, having $x, y \in C_j$ will satisfy *all* the constraints of the form $(x, y)z$ for some $z \notin C_i$. In other words, we may regard M as a clustering over the attributes in A (possibly with some missing attributes), and a good partition P should be such that it agrees with M on the cluster labels of the attributes.

Denote the set of such clusterings as $\mathcal{M}$ = $\{M_1, ..., M_n\}$, where M_i is the clustering given by the schema S_i. Since different schemas may give different clusterings, our problem is a problem of *clustering aggregation*: we seek a partition P such that P maximally agrees with the clusterings in $\mathcal{M}$. Unfortunately, clustering aggregation is also a NP-complete problem [33]. As such, PARTITION implements an approximation algorithm which can be regarded as a variant of the AGGLOMERATIVE algorithm in [33].

Given a set of attributes A and a set of clusterings $\mathcal{M} = \{M_1, ..., M_n\}$, PARTITION proceeds as follows. For every pair of attributes a and b in A, their *potential* of being in the same cluster, denoted as $p(a, b)$, is given by the number of clusterings in $\mathcal{M}$ which place a and b in the *same* cluster, subtracted by the number of clusterings which place a and b in *different* clusters. PARTITION starts by placing each attribute in A in a cluster by itself. It then repeatedly merges two clusters with the largest potential, where the potential of two clusters is given by the *group-average* of the potentials of the attributes in the two clusters. The merging process stops when no two clusters have a *positive* potential.

Example 21 Consider $\mathcal{S} = \{S_u, S_v, S_w\}$, where S_u and S_v are the schemas shown in Figure 5.1.b and 5.1.c respectively, and S_w is homogeneous to S_u (with the same attributes and structure). Thus, $\mathcal{S}$ contains a set A of 15 unique attributes numbered from a to o. The first call to PARTITION with A and $\mathcal{S}$ returns $P = \{\{a, b\}, \{c, d, e, f, g, h, l, m\}, \{i, j, n\}, \{k, o\}\}$. It can be verified that, after recursive applications of PARTITION on each cluster in P, LMax produces the unified schema as shown in Figure 5.1.d. □

5.3 Handling Irregular Interface Schemas

As discussed earlier, some interface schemas may be irregular in that they may not have fully captured the grouping relationship of attributes on the corresponding interfaces. The irregular schemas may greatly affect the performance of PARTITION which assumes that the maximum clusters of each schema S correctly indicate S's preferences on grouping attributes.

To address this challenge, consider again a set of schemas $\mathcal{S}$, some of which may be irregular. A key observation is that we can exploit other schemas in $\mathcal{S}$ to identify the irregularities in the irregular schemas, assuming that not every schema in $\mathcal{S}$ is irregular. Specifically, consider a schema $S_i \in \mathcal{S}$. We observe that if attributes $a, b \in S_i$ appear in different maximum clusters of S_i, but both appear in the same maximum cluster of some other schema $S_j \in \mathcal{S}$, then it is very likely that the grouping relationship between a and b is implicit on the interface of S_i. Motivated by the above observation, we extend LMax based on the concept of *global maximum clusters* defined as follows.

Definition 13 (**Global maximum cluster**) Consider a set of schemas $\mathcal{S} = \{S_1, ..., S_n\}$, where each schema $S_i \in \mathcal{S}$ gives a set of maximum clusters $M_i = \{C_{i_1}, C_{i_2}, ..., C_{i_k}\}$. We say that a maximum cluster $C_{i_j} \in M_i$ is a *global maximum cluster* if it is not a proper subset of any maximum clusters of any other schemas in $\mathcal{S}$. □

We denote the set of global maximum clusters over the schemas in $\mathcal{S}$ as $\mathcal{C}^{\mathcal{S}} = \{C_1^S, C_2^S, ..., C_m^S\}$, and denote the set of unique attributes in the schemas of $\mathcal{S}$ as A. It is important to note that the clusters in $\mathcal{C}^{\mathcal{S}}$ may not form a partition over A, due to the potential structural conflicts among the schemas of $\mathcal{S}$.

Based on the above definition, we modify PARTITION in LMax. Recall that, given a set of attributes A and a set of clusterings $\mathcal{M} = \{M_1, ..., M_n\}$, where M_i is a set of maximum clusters obtained from schema $S_i \in \mathcal{S}$, PARTITION forms a partition over A via clustering aggregation.

The modified PARTITION consists of the following steps: (1) obtain $\mathcal{C}^{\mathcal{S}}$, a set of global maximum clusters; (2) transform $\mathcal{M}$ into a new set of clusterings $\mathcal{M}' = \{M_1', ..., M_n'\}$, where M_i' is obtained from M_i by combining the clusters in M_i which are subsets of the same global maximum cluster in $\mathcal{C}^{\mathcal{S}}$ into one cluster; (3) perform the clustering aggregation with $\mathcal{M}'$ instead.

The LMax algorithm with the new PARTITION is denoted as GMax.

5.4 Incorporating the Ordering

In this section, we extend LMax (and GMax) to produce ordered schemas. Recall that the function NODE(n_1, n_2, ..., n_k) in LMax creates a node r with the nodes n_i's as its children in no particular order. So the key idea is to order the n_i's such that the precedence constraints from the schemas in $\mathcal{S}$ are satisfied as much as possible. In the following, we first describe the proposed ordering algorithm ORDER, then discuss how to incorporate it into LMax.

ORDER: The algorithm ORDER(C, $\mathcal{S}$) takes as input a set of disjoint groups of attributes $C = \{C_1, C_2, ..., C_k\}$ and a set of schemas $\mathcal{S} = \{S_1, ..., S_n\}$, where each $C_i \subset A$ (A is a set of unique attributes over the schemas in $\mathcal{S}$). It orders the groups in C and outputs a sequence C' = $<C'_1, C'_2, ..., C'_k>$, where each C'_i is a different group in C. Groups are ordered such that the precedence constraints in the schemas of $\mathcal{S}$ are satisfied as much as possible. A precedence constraint $x \prec y$ from a schema $S_i \in \mathcal{S}$ is satisfied by the sequence C' if $x \in C'_i$, $y \in C'_j$, and $i \leq j$.

Precedence Graph: ORDER works on a precedence graph G = (V, E), which is a directed weighted graph induced from C and the precedence constraints from the schemas in $\mathcal{S}$. V is a set of k vertices, $\{v_1, v_2, ..., v_k\}$, where each v_i corresponds to a group $C_i \in C$. For every two vertices v_i and v_j in V, there are two directed weighted edges (v_i, v_j) and (v_j, v_i) in E. The weight of an edge (v_i, v_j) indicates the degree of preference on placing attribute group C_i before attribute group C_j, and is computed as follows.

Denote a set of attribute sequences obtained from the schemas in $\mathcal{S}$ as $Q_S = \{q_{s_1}, q_{s_2}, ..., q_{s_n}\}$, where q_{s_i} is the attribute sequence resulted from a pre-order traversal of the schema $S_i \in \mathcal{S}$. For every two attributes, $x \in C_i$ and $y \in C_j$, we compute a precedence score $p(x, y)$ as the number of sequences in Q_S where x appears before y. Then, the weight of (v_i, v_j) is taken to be the group-average of the pairwise precedence scores, i.e., $\frac{\sum_{x \in C_i, y \in C_j} p(x,y)}{|C_i| * |C_j|}$.

In addition, each vertex $v \in G$ has a *potential*, denoted as $d(v)$, which is the sum of the weights of all outgoing edges from v minus the sum of the weights of all incoming edges to v. Intuitively, $d(v)$ indicates v's potential on being placed ahead of other vertices.

Obtain the Ordering: Given the precedence graph G, ORDER employs a variant of the topological sorting algorithm to obtain an ordering of the vertices in G. It proceeds in the following steps: (1) Select the vertex v which has the largest potential. (2) Delete v from G, along with all the edges incident on v. (3) Update

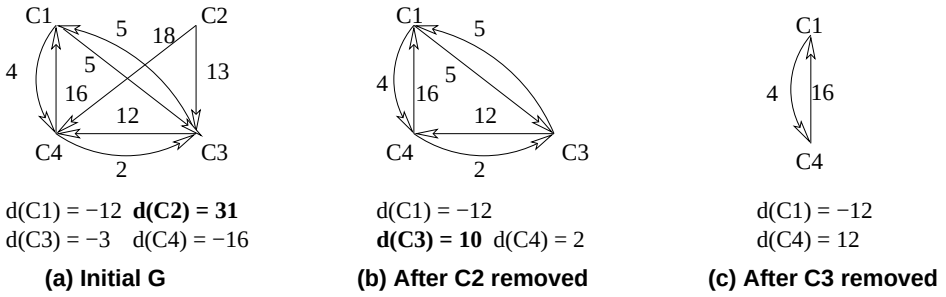


Figure 5.3: An ordering example

the potentials of the remaining vertices. (4) Repeat steps 1–3 until G is empty. (5) Return vertices in the order they are deleted from G.

Example 22 Suppose that $C = \{C_1, C_2, C_3, C_4\}$ and the induced precedence graph G is as shown in Figure 5.3.a, where the potentials of vertices are shown at the bottom. Since C_2 has the largest potential 31, it is first removed from G. The updated graph is shown in Figure 5.3.b. It can be verified that the final ordering is: $<C_2, C_3, C_4, C_1>$. □

***Incorporating* ORDER *into* LMax**: To produce ordered schemas, we replace function NODE(n_1, ..., n_k) in LMax with a new function ORDERNODE(n_1, ..., n_k, $\mathcal{S}$). ORDERNODE proceeds in two steps: (1) Invoke ORDER with $\{C_{n_1}, ..., C_{n_k}\}$ and $\mathcal{S}$ to obtain an ordering: $<C_{n'_1}, ..., C_{n'_k}>$. (2) Create a node r with n'_1, ..., n'_k as the children in this order.

We denote the extended version of LMax and GMax for producing ordered schemas as LMax^o and GMax^o, respectively.

5.5 Empirical Evaluation

We have extensively evaluated LMax and its extensions. The goal of the experiments was to examine if the proposed merging algorithm can produce well-formed unified schemas. In this section, we present the experimental results.

Domain	Leaf Nodes			Internal Nodes			Depth		
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
Airfare	5	15	10.7	1	7	5.1	2	5	3.6
Automobile	2	10	5.1	1	4	1.7	2	3	2.4
Book	2	10	5.4	1	2	1.3	2	3	2.3
Job	3	7	4.6	1	2	1.1	2	3	2.1
Real Estate	3	14	6.7	1	6	2.4	2	4	2.7

Table 5.1: Domains and statistics of the data set

5.5.1 Experiment Setup

Data set: All experiments were performed on the *ICQ* data set [3]. Recall from Sections 2.4 & 3.4 that this data set contains a total of 100 interface schemas in five domains: airfare, auto, book, job, and real estate, with 20 schemas in each domain. Some of relevant statistics of the data set are reproduced in Table 5.1. For each domain, the table shows the minimum, the maximum, and the average number of leaf nodes and internal nodes in the schema trees of that domain. The last three columns show the minimum, the maximum, and the average depth of schema trees in the domain.

Performance metrics: A possible method for measuring the quality of produced unified schemas is to have human assessors examine the schemas and identify the incorrect grouping and ordering among the attributes. Another method, which is more objective, is to compare the unified schema produced by the algorithm with the *optimal* unified schema as defined in Section 5.1. But since finding optimal schemas is computationally expensive, we use alternative metrics **PerSC** and **PerPC** defined below.

(1) **PerSC** is the percentage of *strong* structural constraints from the given set of interface schemas that are satisfied by the unified schema. Intuitively, since conflicting constraints from different schemas can not be satisfied simultaneously, we expect that the optimal schema should satisfy the *strong* constraints. A structural constraint $(x, y)z$ is strong if it appears more often in the given set of interface schemas than its conflicting constraints, $(x, z)y$ and $(y, z)x$. PerSC thus gauges if the unified schema is *structurally* well-formed.

(2) **PerPC** is the percentage of strong *precedence* constraints from the given set of interface schemas that are satisfied by the unified schema. A precedence constraint $a \prec b$ is strong if it appears in the given set of schemas more often than its conflicting constraint $b \prec a$. PerPC thus gauges if the attributes and attribute

Domain	LMax (PerSC%)	GMax (PerSC%)
Airfare	73.6	89.9
Auto	74.0	95.6
Book	91.8	91.8
Job	73.7	89.5
Real Est.	63.6	89.0
Average	75.3	91.2

Table 5.2: The performance of LMax vs. GMax

Domain	LMax^o (PerPC%)	GMax^o (PerPC%)
Airfare	97.3	95.3
Auto	88.7	85.6
Book	96.9	96.9
Job	86.6	84.0
Real Est.	88.8	86.5
Average	91.7	89.7

Table 5.3: The performance of ORDER

groups in the unified schema are intuitively ordered.

Experiments: For each domain, we performed three sets of experiments. The first set of experiments evaluated the performance of LMax and GMax on producing *structurally* well-formed unified schemas, judged by **PerSC** scores. The second set of experiments evaluated the effectiveness of the ordering algorithm ORDER (Section 5.4), measured by **PerPC** scores. The last set of experiments evaluated the sensitivity of the algorithms to the number of input schemas.

5.5.2 Results & Analysis

(1) LMax *vs.* GMax*:* Table 5.2 shows, for each domain, the performance of LMax vs. GMax on integrating all the 20 schemas in that domain. We observe that the PerSC score of GMax increases significantly in four out of five domains, ranging from 15.8% in the job domain to as high as 25.4% in the real estate domain. This highlights the prevalence of irregularities over the interface schemas. Overall, the average PerSC score increases from 75.3% to 91.2%. This indicates the effectiveness of GMax in handling irregularities in interface schemas.

(2) LMax^o *vs.* GMax^o*:* To evaluate the ordering algorithm ORDER, we examined the performance of LMax^o and GMax^o (i.e., LMax and GMax with ORDER incorporated) by their PerPC scores. Table 5.3 shows, for each domain, their performance on integrating all the 20 schemas in that domain. We observe that the PerSC scores range from 84% in the job domain by GMax^{o}and to as high as 97.3% in the airfare domain by LMax^o. This indicates the effectiveness of ORDER.

(3) *Performance sensitivity:* Figure 5.4 shows the performance of GMax and LMax (in terms of their PerSC scores) in the airfare domain when the number of schemas increases from 2 to 20 at a step of 2. We can observe that GMax consistently outperformed LMax when $n > 4$. Furthermore, GMax maintained a very high accuracy rate (about 90%) when $n > 6$.

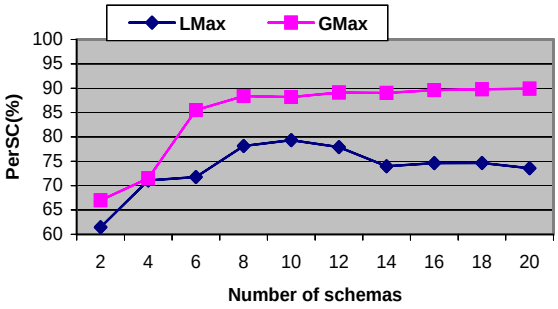


Figure 5.4: The performance of LMax vs. GMax in the **airfare** domain with varied numbers of schemas

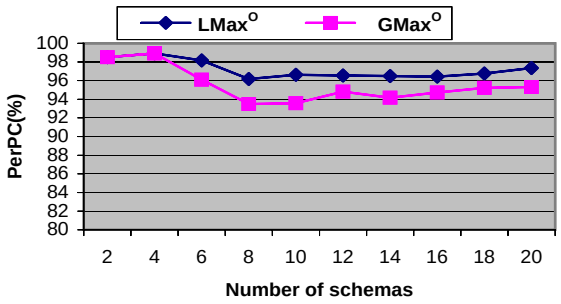


Figure 5.5: The performance of LMax^o vs. GMax^o in the **airfare** domain with varied numbers of schemas

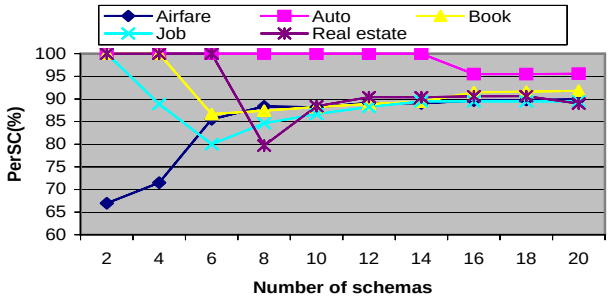


Figure 5.6: The performance of GMax over five domains when the number of schemas varies

Figure 5.5 shows the performance of both LMax^o and GMax^o (in terms of their PerPC scores) in the airfare domain when the number of schemas varies. We can observe that both LMax^o and GMax^o achieved very consistent accuracy rates (above **92%**). Furthermore, the performance of GMax^o is comparable to to that of LMax^o: the largest difference between the two is 3.1 percentage points (when $n = 10$).

Finally, Figure 5.6 shows the performance of GMax over the five domains when the number of schemas varies. We can observe that when $n > 8$, the PerSC score of GMax is maintained at the 90% level for all the domains.

5.5.3 Remarks

Note that we also need to properly annotate the elements in the unified schema. To annotate a leaf element (i.e., an attribute), we use the most frequent label for the attribute over the input schemas. To annotate an internal element n, we use the label of an element m from some input schema S such that the number of common attributes between C_n and C_m (see Section 5.2) is the largest.

Note also that the values of attributes in the unified schema may be obtained by combining the values of similar attributes in the input schemas [43].

Finally, if there is a 1:m mapping between attribute a from schema X and $\{b_1, ..., b_k\}$ from schema Y, we first transform X such that a becomes an internal node in X with b_i's as its children. This is done before the merging process starts.

5.6 Summary

We have described a solution to merging a large scale of diversified interface schemas. The key contributions of the solution are: (a) a novel optimization-based framework for merging schema; (b) an effective merging algorithm based on the idea of clustering aggregation; (c) an effective ordering algorithm for producing ordered schemas; (d) novel performance metrics for measuring the quality of unified schemas.

While our focus was on merging interface schemas, we believe that many of our techniques can be applied to other schema integration tasks, e.g., merging product catalogs from a large number of e-commerce Web sites to produce a master catalog.

Chapter 6

Related Work

In this chapter we review works related to our interface integration solution and discuss in detail how our solution advances the state of the art.

- First, we survey the extensive literature on data integration, discuss an alternative warehousing approach to building data integration systems, and relate our works to the work on integrating unstructured data sources via metasearch engines.
- Second, we discuss related works in schema integration from three aspects: schema extraction, schema matching, and schema merging. We highlight the contributions of our solution to each of these problems.
- Third, we discuss works in Artificial Intelligence and Data Mining that are related to ours, in particular, the works on question-answering and clustering aggregation.

6.1 Data Integration

The problem of integrating a collection of autonomous data sources has been studied in the database community since the early 1980's [92, 83]. Early research was motivated mostly by the need of managing multiple organization-level database systems in a large enterprise or business unit. Integration systems proposed in this context typically come in form of multi-database systems or federated database systems [92]. In the past decade, the increasing number of data sources available on the Web [10, 57] has spurred great interest in building data integration systems to facilitate the retrieval of information from multiple Web sources. Many prototypes have been developed at both academic and industrial research labs. Some of the examples are: TSIMMIS [16], Information Manifold [39], Ariadne [50], CLIDE [78], DISCO [94], Garlic [88], HERMES [4], and Yat [18].

At the high-level, these prototypes share a similar mediator-wrapper architecture as proposed in [101]. Specifically, the integration system consists of a mediator and a set of wrappers, one for each source. The mediator provides a mediated (i.e., global) schema for users to pose queries. For each query, the system first transforms the query into a set of queries on the sources and then sends the queries to the wrappers associated with the respective sources. Next, the wrappers translate the queries into the formats suitable for the sources and execute the queries on the sources.

As discussed earlier, Web data integration is significantly more challenging largely due to two aspects: (1) the *scale* of the problem: for any domain of interests, there may be hundreds or even thousands of data sources on the Web; in contrast, the number of enterprise or business internal data sources is relatively smaller; (2) the *diversity* of the sources: the sources may vary significantly with respect to their content coverage, query capabilities, and response times. As a result, the performance of data integration systems has been recognized as one of the key issues and received a lot of attentions. Many research efforts have focused on two aspects: (1) *Query reformulation* [39, 16, 96]: given a user query, how to translate the query into a set of query plans based on the query capabilities of the sources. (2) *Query optimization* [26, 46, 55, 36]: given a set of query plans, how to identify the best physical execution plan based on the statistics of the sources such as content coverage and query response time.

Another key issue in building these data integration systems is to identify the semantic mappings between the mediated schema and the source schemas. These mappings are crucial for the system to properly reformulate a user query into queries on the sources [25, 22]. We will discuss related works on schema matching in Section 6.2. Note that one of the major differences between these works and ours is that they assume a pre-defined mediated schema while we aim to construct such a schema given a set of sources in a domain of interest.

In order to build integration systems over a collection of Web sources in a domain of interest, we first need to find such sources. Compared to other aspects of data integration systems, the problem of *source discovery* has received little attention. Recently, [7] developed an approach to locating relevant data sources on the Deep Web based on the ideas of focused crawling [14] and reinforcement learning [87].

The sources on the Web may frequently undergo changes, so the maintenance of the integration systems becomes an important issue. One maintenance problem is to detect and repair broken mappings between the mediated schema and the source schemas [73, 68]. Another problem is to verify if the wrapper programs

are still valid when sources undergo changes [58, 52].

Virtual vs. Warehousing: All the above works and ours focus on building a *virtual* integration system [100]. Such an integration system does not store the data itself; rather, all the data are left with the sources. An alternative is to take a "materialized" or warehouse-like approach to integrating a set of data sources [102, 82]. In this approach, all the data from the sources need to be extracted and loaded into a central repository managed by the integration system. Since data do not need to be retrieved on-the-fly from the sources, the warehousing approach can generally achieve a better response time in answering user queries. But the warehousing approach has several key disadvantages:

- *Freshness:* In many application domains, sources may update their contents very frequently. For example, an online bookstore may have books sold out in every minute or may change the prices of the books constantly. For another example, a source on stock quotes may update the prices of stocks at any second. As such, it is very difficult to maintain the freshness of the data stored in the warehouse that come from these sources.

- *Scalability:* As described earlier, there may be hundreds or even thousands of data sources on the Deep Web for any domain of interest and the number is rapidly growing. Further, each data source may contain millions of data records. This sheer amount of data thus poses serious challenges to the warehousing approach which needs to store all the data in a central store (and also keep them up-to-date).

There are typically two methods for obtaining the data from the sources in building a warehouse-like integration system. (1) *Pull:* the system actively fetches the data from the sources. For example, [82] proposes an approach to crawling the Deep Web sources by automatically submitting queries and obtaining results from the sources. (2) *Push:* the sources actively upload their contents to the system. For example, Google permits merchants to upload the information of their products including category, attributes, and data values to its online shopping system [2]. And recently, it introduces a new information gathering system Base [1] which allows users to post contents using a schema provided by the system.

Integrating Unstructured Data Sources: Besides the structured sources considered in our work, the Deep Web also contains a large number of unstructured or text data sources. The problem of integrating

a collection of autonomous text sources has been extensively studied in the context of metasearch engines [34, 65, 111, 108]. Conceptually, a metasearch engine can be regarded as a data integration system over text sources [35].

Building a metasearch engine involves three major tasks which are similar to those in building data integration systems. (1) *Database selection:* Given a user query, the metasearch engine first needs to identify a subset of sources useful for answering the query. To estimate the usefulness of the sources, the engine typically maintains a content summary for each source which may include the document frequency [89] of each word occurring in the documents of the source. [34]. (2) *Query translation:* Different sources may use different query models, e.g., Boolean retrieval model vs. vector space model [89]. So the metasearch engine needs to reformulate the query into queries on the sources based on query models supported by the sources. (3) *Result merging:* Different sources may also use different ranking functions. So the metasearch engine needs to decide on how many documents it should retrieve from the selected sources and how the documents from different sources can be properly merged [108].

6.2 Schema Integration

In this section we relate our work to the existing works on schema integration from three aspects: schema extraction, schema matching, and schema merging.

6.2.1 Schema Extraction

Label Attachment and Schema Extraction: The problem of attaching labels to the attributes on the interfaces and inferring interface schemas has been addressed in several works [47, 82, 113, 44].

[47, 82] propose varied algorithms for attaching labels to the attributes on the interfaces. The algorithms in [47] are mainly based on matching labels with the names of attributes as given in the HTML scripts of the interfaces. Group labeling was also considered, but limited to groups of check boxes or radio buttons. Further, its accuracy rate (80%) is much lower than ours. [82] exploits the spatial location, font size, and font style of labels for label attachment. In contrast, our approach is mainly based on annotation patterns. Group labeling was not considered in [82]. Furthermore, our accuracy rate (95.5% in F1) on attribute labeling is much higher than that in [82], and was achieved on a data set much more complex than that in [82].

[113] focuses on extracting query conditions from interfaces. The query conditions may indicate a

restricted form of attribute groups, e.g., a text input box for author may be associated with a group of radio buttons specifying if the required input is first name, last name, or full name. Such specific grouping of attributes may be handled by a grouping pattern in our structure extraction algorithm. In addition, our accuracy rate in attribute grouping (92.3% in F1) is much higher than 85% in [113]. A more recent work [44] proposes a layout-expression-based approach to extracting attributes and their labels from the interface. Its accuracy rates are comparable to ours. But similar to other existing solutions, its key limitation is that it considers interfaces to be "flat" and therefore fails to obtain important grouping relationships among the attributes from the interfaces.

Wrapper Induction: In addition, our work is related to the works on wrapper induction [53, 21, 6] which also attempt to extract relevant information from Web pages. The key difference is that these works focus on extracting information from result pages returned from the sources rather than from source query interfaces which have very different characteristics.

6.2.2 Schema Matching

Schema matching is one of the key tasks in data integration [25]. It is also a fundamental problem in many other applications [83, 11, 37, 29]. For example, in data warehouse [11], where data from disparate sources need to be transformed into the warehouse format for analysis; in peer data management [37], where semantic mappings among peers' individual schemas are required for the exchange of data among peers; and in personal information management [29], where the semantic association between data items from different applications is needed for the effective searching and browsing of the information on one's desktop.

Schema matching is a well-known challenging problem which has been studied for a few decades [9, 92, 83, 27]. There is also a large body of recent works [22, 24, 25, 38, 41, 42, 43, 48, 59, 64, 70, 72, 80, 98]. Among them, [41, 42, 43, 98] focus on the problem of matching interface schemas. But despite these efforts, schema matching remains as a very difficult problem. In the following, we compare our schema matching solution with existing solutions from several perspectives and highlight our contributions.

Scale: Most of the early matching algorithms typically handle two schemas at a time [83]. So in order to match more than two schemas, we need to proceed in a pairwise fashion. For example, to match three schemas S_1, S_2, and S_3, we need to match three pairs of schemas: (S_1, S_2), (S_1, S_3), and (S_2, S_3). A major

drawback of this pairwise approach to matching a large number of schemas is that schemas are matched in isolation. As we observed earlier, matching attributes may look very different themselves (thus difficult to match) but may be related with the help of similar attributes from other schemas. Motivated by this observation, we have developed an effective clustering-based matching algorithm for large-scale schema matching. The key contributions of our solution include: (1) the novel formulation of schema matching as a clustering problem; and (2) the extensive study of the bridging effect in matching a large number of schemas. The bridging effect for 1:1 mappings is discussed in Section 3.1. The bridging effect for 1:m mappings is discussed in Section 3.2.3, where 1:1 mappings obtained from the clustering process are exploited to discover additional 1:m mappings.

The bridging effect can be related to the idea of mapping reuse [83], where past identified mappings are exploited to assist in the current schema matching task. The key difference is that, in mapping reuse, the mappings which serve as the bridges come from previous matching tasks, while the bridges in our solution are from the input schemas of the matching algorithm.

The bridging effect can also be related to the works in [63, 112]. Specifically, [63] proposes to use a corpus of schemas and mappings in the application domain to help match schemas. [112] utilizes a unlabeled text corpus to connect new examples with labeled examples to achieve a similar bridging effect in the context of text classification.

Finally, [41] also attempts to achieve a better matching accuracy by matching a large number of schemas at once. It proposed an approach to learning a schema model by exploiting the occurrence statistics of attribute labels over a collection of query interfaces.

Mapping Cardinality: Most of the current works on schema matching only consider 1:1 mappings [83, 24, 25]. [64, 70] also handles 1:m mappings but the techniques utilized are completely different from ours. The average matching accuracy reported in [70] (52%) is substantially worse than our accuracy rate although the test data are different. And since [64] only reports the comparison of their approach with several other systems on two example schemas, it is not clear how their system performs on a large data set.

[41, 43] also study the problem of matching interface schemas. While 1:m mappings frequently occur among the attributes on the interfaces as we have observed, both of them only considered 1:1 mappings. Furthermore, both of them model the interfaces with flat schemas and do not utilize the ordering and sibling relationships of the attributes and the hierarchical structure of the interfaces, which are highly valuable in

improving the matching accuracy as we have shown.

[42] addressed the problem of finding complex mappings among interface schemas by exploiting the co-occurrence statistics of attributes over interface schemas. [22] proposes iMap, a semi-automatic matching system which employs a set of searchers to discover both 1-1 and complex mappings. Each searcher focuses on finding a different type of candidate mappings. The system also exploits past identified mappings to help improve matching accuracy.

Data Instances: Many existing matching solutions do not exploit data instances [41, 42, 64]. Nevertheless, as we have shown, the instances of attributes are very important evidences on the semantics of the attributes and can be exploited to significantly improve the matching accuracy.

The importance of instances in schema matching has also been observed in [25, 43]. In particular, [25] proposes a machine-learning approach to matching the mediated schema with the schemas of data sources. It learns naive Bayes instance classifiers from training examples and employs the classifiers to discover the mappings. In contrast, our solution does not require a set of manually prepared training examples. Our utilization of instances in suggesting possible mappings resembles the value correspondence problem in [72].

[43, 98] also exploit instances for matching interface schemas. But neither of them addresses the problem of lacking data instances on query interfaces.

User Interaction and Parameter Learning: User interaction is an effective way to improve the system performance and has been successfully employed in many areas, including information retrieval [89, 91], information extraction [51], and data cleaning [85, 90]. Nevertheless, there is little work on utilizing user interaction in schema matching tasks [83].

In [25], users provide feedback on the system identified mappings. The feedback is captured as constraints which are then incorporated into the matching system and utilized in future matching tasks. In contrast, in our approach, users interact to resolve the uncertain mappings *during* the current matching process. [69] proposes a mass collaboration approach to schema matching, where the system initially deployed contains only a partial set of mappings and the users of the system are leveraged to help determine the rest of the mappings.

Typically, matching systems have a set of parameters to be tuned in order to yield good performance

in a specific domain [41, 43]. And the tuning is often done in a trial-and-error fashion, without any principled guidance. In contrast, we have developed an effective approach to learning the system parameters, particularly the matching thresholds, with a limited amount of user interaction. Thresholding function can be regarded as a linear classifier. The learning of classifiers has been extensively studied in the machine learning literature [74]. The interactive learning of classifiers was explored in [90] in the context of record deduplication. Our approach to the interactive learning of thresholds is similar to [90] in the goal of reducing the amount of user interaction, but our application area, namely schema matching, is different. Our experiments show that our approach can effectively shrink the confusion region with a small amount of user interaction.

6.2.3 Schema Merging

The problem of merging a set of schemas into a global schema has been studied in the database community in the context of view integration and database integration [9, 92, 77]. In view integration, a global conceptual schema of a database needs to be constructed from a set of user defined views; in database integration, a set of existing databases need to be integrated into a single distributed database.

In these works, conflicts among schemas are typically resolved through manual conformation or aligning the input schemas prior to the merging step. Further, as [9] indicates, none of these approaches provides an analysis or proof on the correctness of the unified schemas.

[80] studies schema merging in the context of model management. It categorizes the conflicts among schemas into several types: representation (including naming and structural), meta-model, and fundamental. It proposes an algorithm which automatically resolves fundamental conflicts to ensure the unified schema is indeed a model (e.g. tree).

As discussed earlier, compared to schema matching [41, 43, 98, 107], there has been little work on merging interface schemas. [43] reports some preliminary work on schema merging and proposes several simple heuristics to order the attributes on the global interface. But similar to other works, it considers the interfaces to be flat, thus does not produce well-structured interfaces.

6.3 Artificial Intelligence & Data Mining

Question-Answering & Information Extraction: Question answering has been an active research area in both AI and IR communities (e.g., [54, 81, 84]). Our approach of gathering instances from the Surface Web is motivated in part by the works on Web-based question answering such as AskMSR [13] and Mulder [54]. In particular, similar to Mulder and AskMSR, we also exploit the idea of "redundancy-based extraction", where the scale and the redundancy of the information on the Surface Web are leveraged to obtain answers to questions from simple sentences, whose syntaxes are relatively easy to analyze.

There are also many works on Information Extraction in the AI community [17, 31]. Many of them rely on supervised learning techniques to train the systems, while our approach of training instance classifiers for interface attributes is fully automatic.

Our approach of gathering instances from the Web is also inspired by the works on exploiting the Web to populate ontologies, such as KnowItAll [31]. But the task of gathering instances for interface attributes is significantly more challenging as we have discussed. Furthermore, we believe that the techniques we developed for gathering instances of interface attributes such as label syntax analysis and outlier detection, can also be incorporated into the Web-based information extraction systems such as [17, 31].

Clustering Aggregation: Clustering is an effective approach to discovering natural groups among a set of objects and has been extensively studied in Data Mining [40]. [33] studies clustering aggregation in a general setting, where the problem is to find a clustering which agrees as much as possible with a given set of clusterings. It shows that the clustering aggregation approach can produce a better clustering on a set of objects than the traditional clustering algorithms. In a sense, our approach of combining schema trees via recursive applications of clustering aggregation can be regarded as one way of aggregating hierarchical clusterings, such as dendrograms and classification trees.

Similar aggregation idea has also been employed to build consensus trees in Genetics and Molecular Biology [76]. The consensus tree problem is formulated as follows: given a set S of candidate evolutionary trees for a set of species, find a consensus tree T such that T maximally agrees with the trees in S. The intuition is that by combining the information from different and possibly partially disjoint trees, such a consensus tree may better represent the true evolutionary relationships among the species.

Machine Learning: Naive Bayes classifier has been extensively studied in Machine Learning [74] and shown to be very effective in varied application domains such as text classification [28]. Several recent learning-based schema matching solutions [25, 22, 83] also employ the naive Bayes algorithm for training instance classifiers. Typically these classifiers treat instances as bags of tokens [67]. In other words, each instance may be represented as a feature-value vector where each feature corresponds to a token and the value of the feature is the number of occurrences of the token in the instance. Training such classifiers often require a large number of training examples so that the occurrence probabilities of the tokens can be reliably estimated.

In comparison with these conventional classifiers, the naive Bayes instance classifier proposed in Section 4.2 has several distinct aspects. First, its features are based on validation phrases instead of tokens. In other words, it characterizes an instance (e.g. New York) by its co-occurrence statistics with the attribute label (e.g., city) over the Web, via a set of validation phrases (e.g., cities such as). Second, the values of its features are based on the hit counts of validation queries (e.g., cities such as New York) for the instance instead of the number of occurrences of the tokens in the instance. Third, the number of its features is the same as the number of validation phrases (typically < 10) vs. the number of tokens (often hundreds or even thousands) in the conventional classifiers. Fourth, it does not require a large number of training examples, rather it relies on the statistics readily available from the Web.

Chapter 7

Conclusion

The problem of integrating Deep Web data sources is becoming increasingly crucial. An important first step in building integration systems on the Deep Web is the integration of source query interfaces. In this dissertation, we have presented a novel and effective interface integration system IceQ. In this chapter, we summarize its key contributions and discuss directions for future research.

7.1 Key Contributions

The integration of source query interfaces involves three major tasks: *schema extraction*, *schema matching*, and *schema extraction*. As discussed earlier, each of these tasks is very challenging in its own right, and existing solutions suffer from many limitations. To address these limitations, we have made several key contributions.

- We proposed to use ordered-tree schema for modeling query interfaces. Such a schema has been shown to be well suited for representing the structure of query interfaces. We presented a novel schema extraction algorithm which captures the grouping relationships of attributes by performing spatial clustering on the attributes.

- We developed a novel clustering-based schema matching algorithm to accurately identify attribute mappings over a large number of schemas. We also put the human integrators back in the loop and presented several approaches to effectively learning matching thresholds and resolving uncertain mappings with a small amount of user interaction.

- We addressed the problem of lacking attribute instances and developed an approach to gathering instances from the Web via novel adaptations of question-answering techniques commonly employed in Artificial Intelligence. We further demonstrated that learned instances can be exploited to significantly

boost the matching accuracy.

- We proposed a novel optimization framework for merging schemas. Within this framework, we developed an effective merging algorithm based on the idea of clustering aggregation. We also proposed several novel metrics for measuring the quality of constructed unified schemas.

- We have conducted extensive experiments over varied real-world domains to evaluate our solutions. And the results indicate that the proposed solutions are very effective and significantly outperform existing solutions.

7.2 Future Directions

As discussed earlier, interface integration is a first step towards the virtual integration of Deep Web sources. To achieve a "deeper" integration of Deep Web sources beyond their query interfaces, we are facing several additional challenging tasks, including *query optimization*, *result merging*, and *system maintenance*. In this section, we discuss several directions for future work. Section 7.2.1 describes the important problem of learning the characteristics of data sources such as content coverage for query optimization. Section 7.2.2 proposes to incorporate online user feedback during query execution. And Section 7.2.3 describes the problem of merging results from different sources and our preliminary work on extracting relevant contents from result pages.

7.2.1 Learning Characteristics of Sources

There may be hundreds or even thousands of sources in a domain of interest and these sources may vary greatly in their coverage of domain contents. For example, a car-sale source may carry a large number of Ford's but only a few Honda's. Furthermore, sources may vary significantly in their query response time, depending on their computation resources and networking capacities. It is thus crucial to understand the characteristics of different sources [75], so that efficient and effective query plans over the sources can be formulated (see also the second direction below).

An interesting problem here is that although our current work focuses on complex interfaces, some Deep Web sources have simple keyword-based search interfaces. Note that these sources still contain structured contents. So the question is how to design *effective* (keyword-based) probing queries to these sources, in

order to discover the content coverage of the sources? The key issue would be not to overwhelm the sources with a large number of probing queries [102].

7.2.2 Incorporating Online User Feedback in Query Execution

For each query users pose on the global query interface, the system needs to reformulate the query into a set of query plans. Each query plan specifies a way of accessing a set of sources and combining the results from the sources. Given a multitude of diversified sources, there may be a large number of possible query plans for a global query. And different plans typically may have quite different utilities, e.g., in terms of content coverage and execution time. So one of the key challenges is how to efficiently order the query plans so that better plans get executed first [26].

Another major challenge, we believe, is how to effectively incorporate user feedback *during* the query execution. The key idea would be to put the users in the loop: rather than executing a set of query plans to the end (which may take a long time), users may observe the results from the executed plans and immediately indicate the relevancy of the results to the system. Based on the feedback from the users, the system may either modify, e.g., by re-ordering, the remaining plans, or generate a new set of query plans which are deemed to be more likely to produce relevant results.

Relevance feedback has been exploited and shown to be very effective in other contexts, e.g., information retrieval [91] and schema matching [107]. But the idea of employing relevance feedback in query reformulation is relatively new and has received little attention. We will investigate the techniques from active learning [95] and online query processing [86] to extensively explore this direction.

7.2.3 Merging Search Results

In order to provide the users with a unified result interface, we need to properly merge the results from different sources. Typically, each source returns its results in a set of Web pages, each containing a list of data records which meet the query conditions. But the problem is that, similar to query interfaces, result pages do not specify where to find data records on the pages and how to identify the columns of each record. Furthermore, result pages may often contain additional information other than data records, e.g., advertisements and navigation links. So the challenge is how to automatically identify and extract data records from the result pages.

The problem of extracting information from semi-structured texts, such as result pages dynamically generated by the Deep Web sources, is being actively researched. The current state-of-art extraction systems (e.g., [21, 6]) are still very fragile. In [105], we proposed a novel knowledge-driven approach to extracting data records from result pages. The key idea is to exploit the initial knowledge learned from query interfaces on domain concepts and instances to accurately identify data regions and data records on the result pages. Preliminary experiments indicated the feasibility of the approach. But the effectiveness of the approach on more complex result pages needs further more extensive investigations.

The contents of different sources may often overlap, so the second issue is how to accurately identify duplicate data records from different sources. Furthermore, since the results combined from multiple sources may contain hundreds or thousands of data records, another issue is how to properly rank the data records by their degree of relevancy, so that more relevant records may be presented to the users first.

In summary, I believe that Deep Web data integration is an urgent and yet extremely rewarding research problem due to the phenomenal growing rate of the contents on the Deep Web. Besides continuing the research on the integration of Deep Web sources along the directions outlined above, I plan to also extend the research to building data integration systems over a greater variety of data sources, including unstructured data sources on the Deep Web as well as business and enterprise internal data sources. To gain first-hand knowledge on the problem, I intend to partner with researchers and e-commerce companies in the application domain. I strongly believe that these, together with my extensive interdisciplinary experiences on database system [108, 111, 60, 30], information retrieval [108, 109, 110, 71, 61], and machine learning [104, 103, 107, 106], will lead to a very fruitful research.

References

[1] Base. http://base.google.com/.

[2] Froogle. http://froogle.google.com/.

[3] UIUC Web integration repository. http://metaquerier.cs.uiuc.edu/repository/.

[4] S. Adali, K. Candan, Y. Papakonstantinou, and V. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD'96)*, pages 137–146, 1996.

[5] A. Aho, Y. Sagiv, T. Szymanski, and J. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM Journal on Computing*, 10(3):405–421, 1981.

[6] A. Arasu and H. Garcia-Molina. Extracting structured data from Web pages. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, pages 337–348, 2003.

[7] L. Barbosa and J. Freire. Searching for hidden-Web databases. In *Proceedings of the 8th ACM SIGMOD International Workshop on Web and Databases (WebDB'05)*, pages 1–6, 2005.

[8] V. Barnett and T. Lewis. *Outliers in Statistical Data*. John Wiley & Sons, 1994.

[9] C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[10] M. Bergman. The Deep Web: Surfacing the hidden value. *BrightPlanet.com (http://www.brightplanet.com/technology/deepweb.asp)*, 2000.

[11] P. Bernstein and E. Rahm. Data warehouse scenarios for model management. In *Proceedings of International Conference on Entity-Relationship Modeling*, pages 1–15, 2000.

[12] E. Brill. Some advances in rule-based part of speech tagging. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, volume 1, pages 722–727, 1994.

[13] E. Brill, S. Dumais, and M. Banko. An analysis of the AskMSR question-answering system. In *Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing (EMNLP'02)*, pages 257–264, 2002.

[14] S. Chakrabarti, M. van den Berg, and B. Dom. Focused crawling: A new approach to topic-specific Web resource discovery. In *Proceeding of the 8th International Conference on World Wide Web (WWW'99)*, pages 1623–1640.

[15] K. Chang, B. He, C. Li, M. Patel, and Z. Zhang. Structured databases on the Web: Observations and implications. *ACM SIGMOD Record*, 33(3):61–70, 2004.

[16] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proceedings of the 10th Meeting of the Information Processing Society of Japan (IPSJ'94)*, pages 7–18, 1994.

[17] P. Cimiano, S. Handschuh, and S. Staab. Towards the self annotating web. In *Proceedings of the 13th International Conference on World Wide Web (WWW'04)*, pages 462–471, 2004.

[18] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*, pages 177–188, 1998.

[19] W. Cohen. Data integration using similarity joins and a word-based information representation language. *ACM Transactions on Information Systems (TOIS)*, 18(3):288–321, 2000.

[20] W. Cohen, R. Schapire, and Y. Singer. Learning to order things. *Journal of Artificial Intelligence Research (JAIR)*, 10:243–270, 1999.

[21] V. Crescenzi, G. Mecca, and P. Merialdo. RoadRunner: Towards automatic data extraction from large Web sites. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 109–118, 2001.

[22] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. iMAP: Discovering complex semantic matches between database schemas. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 383–394, 2004.

[23] L. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[24] H. Do and E. Rahm. Coma – A system for flexible combination of schema matching approaches. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, pages 610–621, 2002.

[25] A. Doan, P. Domingos, and A. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD'01)*, pages 509–520, 2001.

[26] A. Doan and A. Halevy. Efficiently ordering query plans for data integration. In *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE'02)*, page 393, 2002.

[27] A. Doan and A. Halevy. Semantic integration research in the database community: A brief survey. *AI Magazine, Special Issue on Semantic Integration*, 26(1):83–94, 2005.

[28] P. Domingos and M. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2-3):103–130, 1997.

[29] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD'05)*, pages 85–96, 2005.

[30] E. Dragut, W. Wu, P. Sistla, C. Yu, and W. Meng. Merging source query interfaces on Web databases. In *Proceedings of the 22nd IEEE International Conference on Data Engineering (ICDE'06)*, page 46, 2006.

[31] O. Etzioni, M. Cafarella, et al. Web-scale information extraction in KnowItAll. In *Proceedings of the 13th International Conference on World Wide Web (WWW'04)*, pages 100–110, 2004.

[32] C. Fellbaum, editor. *WordNet: An On-Line Lexical Database and Some of its Applications*. MIT Press, Cambridge, MA, 1998.

[33] A. Gionis, H. Mannila, and P. Tsaparas. Clustering aggregation. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 341–352, 2005.

[34] L. Gravano and H. García-Molina. Generalizing GlOSS to vector-space databases and broker hierarchies. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB'95)*, pages 78–89, 1995.

[35] L. Gravano and Y. Papakonstantinou. Mediating and metasearching on the internet. *IEEE Data Engineering Bulletin*, 21(2):28–36, 1998.

[36] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 276–285, 1997.

[37] A. Halevy, Z. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The PIAZZA peer data management system. *IEEE Transactions Knowledge and Data Engineering (TKDE)*, 16(7):787–798, 2004.

[38] A. Halevy and J. Madhavan. Corpus-based knowledge representation. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 1567–1572, 2003.

[39] A. HaLevy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB'96)*, pages 251–262, 1996.

[40] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2006.

[41] B. He and K. Chang. Statistical schema matching across Web query interfaces. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, pages 217–228, 2003.

[42] B. He, K. C. Chang, and J. Han. Discovering complex matchings across Web query interfaces: A correlation mining approach. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'04)*, pages 148–157, 2004.

[43] H. He, W. Meng, C. Yu, and Z. Wu. WISE-Integrator: An automatic integrator of Web search interfaces for e-commerce. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 357–368, 2003.

[44] H. He, W. Meng, C. Yu, and Z. Wu. Constructing interface schemas for search interfaces of Web databases. In *Proceedings of the 6th International Conference on Web Information Systems Engineering (WISE'05)*, pages 29–42, 2005.

[45] M. Hearst. Automatic acquisition of hyponyms from large text corpora. In *Proceedings of the 14th International Conference on Computational Linguistics*, pages 539–545, 1992.

[46] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD'99)*, pages 299–310, 1999.

[47] O. Kaljuvee, O. Buyukkokten, H. Garcia-Molina, and A. Paepcke. Efficient Web form entry on PDAs. In *Proceedings of the 10th International Conference on World Wide Web (WWW'01)*, pages 663–672, 2001.

[48] J. Kang and J. Naughton. On schema matching with opaque column names and data values. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, 2003.

[49] L. Kaufman and P. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, 1990.

[50] C. Knoblock, S. Minton, J. Ambite, N. Ashish, I. Muslea, A. Philpot, and S. Tejada. The ariadne approach to Web-based information integration. *International Journal of Cooperative Information Systems*, 10(1-2):145–169, 2001.

[51] T. Kristjannson, A. Culotta, P. Viola, and A. McCallum. Interactive information extraction with constrained conditional random fields. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI'04)*, pages 412–418, 2004.

[52] N. Kushmerick. Wrapper verification. *World Wide Web*, 3(2):79–94, 2000.

[53] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 729–737, 1997.

[54] C. Kwok, O. Etzioni, and D. Weld. Scaling question answering to the Web. In *Proceedings of the 10th International Conference on World Wide Web (WWW'01)*, pages 150–161, 2001.

[55] E. Lambrecht, S. Kambhampati, and S. Gnanaprakasam. Optimizing recursive information-gathering plans. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 1204–1211, 1999.

[56] J. Larson, S. Navathe, and R. Elmasri. A theory of attribute equivalence in databases with application to schema integration. *IEEE Transactions on Software Engineering*, 15(4):449–463, 1989.

[57] S. Lawrence and C. Giles. Accessibility of information on the Web. *Nature*, 400:107–109, July 1999.

[58] K. Lerman, S. Minton, and C. Knoblock. Wrapper maintenance: A machine learning approach. *Journal of Artificial Intelligence Research (JAIR)*, 18:149–181, 2003.

[59] W. Li and C. Clifton. Semint: A tool for identifying attribute correspondence in heterogeneous databases using neural networks. *Data & Knowledge Engineering*, 33(1):49–84, 2000.

[60] K. Liu, C. Yu, W. Meng, W. Wu, and N. Rishe. A statistical method for estimating the usefulness of text databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(6):1422–1437, 2002.

[61] S. Liu, C. Yu, and W. Wu. UIC at TREC 2002: Web track. In *Proceedings of the 2002 Text Retrieval Conference (TREC'02)*, 2002.

[62] L. Lovasz and M. Plummer. *Matching Theory*. North-Holland, Amsterdam, 1986.

[63] J. Madhavan, P. Bernstein, A. Doan, and A. Halevy. Corpus-based schema matching. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 57–68, 2005.

[64] J. Madhavan, P. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 49–58, 2001.

[65] B. Magnini, M. Negri, and H. Tanev. Is it the right answer? Exploiting Web redundancy for answer validation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL'02)*, pages 425–432, 2002.

[66] C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.

[67] A. McCallum and K. Nigam. A comparison of event models for naive Bayes text classification. In *AAAI-98 Workshop on Learning for Text Categorization*, 1998.

[68] R. McCann, B. AlShelbi, Q. Le, H. Nguyen, L. Vu, and A. Doan. Maveric: Mapping maintenance for data integration systems. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB'05)*, pages 1018–1030, 2005.

[69] R. McCann, A. Doan, A. Kramnik, and V. Varadarajan. Building data integration systems via mass collaboration. In *Proceedings of the SIGMOD-03 Workshop on the Web and Databases (WebDB'03)*, pages 25–30, 2003.

[70] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, pages 117–128, 2002.

[71] W. Meng, K. Liu, C. Yu, W. Wu, and N. Rishe. Estimating the usefulness of search engines. In *Proceedings of the 15th International Conference on Data Engineering (ICDE'99)*, pages 146–153, 1999.

[72] R. Miller, L. Haas, and M. Hernandez. Schema mapping as query discovery. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'00)*, pages 77–88, 2000.

[73] Y. Velegrakis R. Miller and L. Popa. Mapping adaptation under evolving schemas. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 584–595, 2003.

[74] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[75] Z. Nie, S. Kambhampati, and U. Nambiar. Effectively mining and using coverage and overlap statistics for data integration. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 17(5):638–651, 2005.

[76] Anna Östlin. *Constructing Evolutionary Trees: Algorithms and Complexity*. PhD thesis, Lund University, Sweden, 2001.

[77] C. Parent and S. Spaccapietra. Issues and approaches of database integration. *Communications of the ACM*, 41(5):166–178, 1998.

[78] M. Petropoulos, A. Deutsch, and Y. Papakonstantinou. Interactive query formulation over Web service-accessed sources. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*, pages 253–264, 2006.

[79] M. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[80] R. Pottinger and P. Bernstein. Merging models based on given correspondences. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 826–873, 2003.

[81] D. Radev, H. Qi, Z. Zheng, S. Blair-Goldensohn, Z. Zhang, W. Fan, and J. Prager. Mining the Web for answers to natural language questions. In *Proceedings of the Tenth International Conference on Information and Knowledge Management (CIKM'01)*, pages 143–150, 2001.

[82] S. Raghavan and H. Garcia-Molina. Crawling the hidden Web. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 129–138, 2001.

[83] E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *The International Journal on Very Large Data Bases (VLDB Journal)*, 10(4):334–350, 2001.

[84] G. Ramakrishnan, S. Chakrabarti, D. Paranjpe, and P. Bhattacharya. Is question answering an acquired skill? In *Proceedings of the 13th International Conference on World Wide Web (WWW'04)*, pages 111–120, 2004.

[85] V. Raman and J. Hellerstein. Potter's wheel: An interactive data cleaning system. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 381–390, 2001.

[86] V. Raman and J. Hellerstein. Partial results for online query processing. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*, pages 275–286, 2002.

[87] J. Rennie and A. McCallum. Using reinforcement learning to spider the Web efficiently. In *Proceedings of the 16th International Conference on Machine Learning (ICML'99)*, pages 335–343, 1999.

[88] M. Roth and P. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 266–275, 1997.

[89] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McCraw-Hill, New York, 1983.

[90] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'02)*, pages 269–278, 2002.

[91] X. Shen and C. Zhai. Active feedback in ad hoc information retrieval. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'05)*, pages 59–66, 2005.

[92] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys (CSUR)*, 22(3):183–236, 1990.

[93] S. Tejada, C. Knoblock, and S. Minton. Learning object identification rules for information integration. *Information Systems*, 26(8):607–633, 2001.

[94] A. Tomasic, O. Kapitskaia, H. Naacke, P. Bonnet, L. Raschid, and R. Amouroux. The distributed information search component (Disco) and the World Wide Web. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD'97)*, pages 546–548, 1997.

[95] S. Tong. *Active Learning: Theory and Applications*. PhD thesis, Stanford University, 2001.

[96] J. Ullman. Information integration using logical views. In *Proceedings of the 6th International Conference on Database Theory (ICDT'97)*, pages 19–40, 1997.

[97] C. van Rijsbergen. *Information Retrieval*. Butterworths, London, 1979.

[98] J. Wang, J. Wen, F. Lochovsky, and W. Ma. Instance-based schema matching for Web databases by domain-specific query probing. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB'04)*, pages 408–419, 2004.

[99] J. Wang and K. Zhang. Finding similar consensus between trees: An algorithm and a distance hierarchy. *Pattern Recognition*, 34:127–137, 2001.

[100] J. Widom. Integrating heterogeneous databases: lazy or eager? *ACM Computuing Surveys (CSUR)*, 28(4), 1996.

[101] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25:38–49, 1992.

[102] P. Wu, J. Wen, H. Liu, and W. Ma. Query selection techniques for efficient crawling of structured Web sources. In *Proceedings of the 22nd IEEE International Conference on Data Engineering (ICDE'06)*, page 47, 2006.

[103] W. Wu, A. Doan, and C. Yu. Merging interface schemas on the Deep Web via clustering aggregation. In *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM'05)*, pages 801–804, 2005.

[104] W. Wu, A. Doan, and C. Yu. WebIQ: Learning from the Web to match Deep-Web query interfaces. In *Proceedings of the 22nd IEEE International Conference on Data Engineering (ICDE'06)*, page 44, 2006.

[105] W. Wu, A. Doan, C. Yu, and W. Meng. Bootstrapping domain ontology for Semantic Web services from source Web sites. In *Proceedings of the 6th VLDB Workshop on Technologies for E-Services (VLDB-TES'05)*, pages 11–22, 2005.

[106] W. Wu, A. Doan, C. Yu, and W. Meng. Towards a highly effective integration of source query interfaces on the Deep Web. *Submitted to ACM Transactions on Database Systems (TODS)*, 2006.

[107] W. Wu, C. Yu, A. Doan, and W. Meng. An interactive clustering-based approach to integrating source query interfaces on the Deep Web. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 95–106, 2004.

[108] W. Wu, C. Yu, and W. Meng. Database selection for longer queries. In *Proceedings of the 2004 Meeting of International Federation of Classification Societies*, 2004 (Invited paper).

[109] C. Yu, K. Liu, W. Wu W. Meng, and N. Rishe. Finding the most similar documents across multiple text databases. In *Proceedings of the 1999 IEEE Conference on Advances in Digital Libraries (ADL'99)*, pages 150–162, 1999.

[110] C. Yu, W. Meng, K. Liu, W. Wu, and N. Rishe. Efficient and effective metasearch for a large number of text databases. In *Proceedings of the 8th ACM International Conference on Information and Knowledge Management (CIKM'99)*, pages 217–224, 1999.

[111] C. Yu, W. Meng, W. Wu, and K. Liu. Efficient and effective metasearch for text databases incorporating linkages among documents. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD'01)*, pages 187–198, 2001.

[112] S. Zelikovitz and H. Hirsh. Improving short-text classification using unlabeled background knowledge to assess document similarity. In *Proceedings of the 17th International Conference on Machine Learning (ICML'00)*, pages 1183–1190, 2000.

[113] Z. Zhang, B. He, and K. Chang. Understanding Web query interfaces: Best-effort parsing with hidden syntax. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 107–118, 2004.

Author’s Biography

Wensheng Wu grew up in Fuqin, a city of Fujian province in the southern part of China. After he finished his high school in Fuqin No. 1 Middle School in 1986, he moved to Shanghai, the largest city of China. There he earned his B.E. degree from Tongji University in 1991, and his M.S. degree from Fudan University in 1994. He came to US in 1996, originally studied at the University of Illinois at Chicago. In 1999, he moved to the University of Illinois at Urbana-Champaign with his family and earned his Ph.D. there in 2006. All of his hard-earned degrees are in Computer Science.